Specification, Model Generation, and Verification of Distributed Applications

Eric Madelaine

UNIVERSITÉ DE NICE SOPHIA ANTIPOLIS

Specification, Model Generation, and Verification of Distributed Applications

Mémoire de Synthèse présenté à l'Université de Nice Sophia Antipolis
pour l'obtention d'une

HABILITATION À DIRIGER LES RECHERCHES
Spécialité Informatique

par

Eric Madelaine

Soutenue le 29 septembre 2011
devant la commission d'examen composée de MM. :

Président :	Pr. Gérard Berry	INRIA Sophia Antipolis - Méditérranée
Rapporteurs :	Pr. Frantisek Plasil	Charles University, Prague
	Pr. Christian Attiogbé	Université de Nantes
	M. Radu Mateescu	INRIA Grenoble - Rhônes-Alpes
Examinateurs :	Pr. Denis Caromel	Université de Nice Sophia-Antipolis
	Pr. Elie Najm	Télécom ParisTech, Paris

Mes remerciements vont naturellement à mes collègues, au sein de l'équipe Oasis, de l'INRIA à Sophia-Antipolis, mais aussi de bien plus loin. Un travail de recherche de ce type est avant tout un travail d'équipe, et le soutien de tous, au quotidien dans les affres d'une soumission d'article ou de proposition de projet, ou au hasard des rencontres dans une conférence à l'autre bout du monde, fait de nous un peu plus qu'un chercheur solitaire en tête à tête avec son ordinateur. J'ai aujourd'hui une pensée particulière pour Isabelle Attali qui avait créé notre équipe, avait su lui insuffler une dynamique remarquable, et nous a quitté tragiquement en décembre 2004.

Mes remerciements vont aussi aux doctorants, qui ont eu une place centrale dans ce travail, Didier, Valérie chez Meije, Tomás, Rabéa, Antonio chez Oasis, mais aussi à tous les étudiants qui ont apporté leur brique à notre édifice, leur enthousiasme et leur convivialité.

C'est avec grand plaisir que je remercie très sincèrement mes rapporteurs et tous les membres de mon jury d'Habilitation, pour ce temps précieux qu'ils ont bien voulu consacrer à l'évaluation de mes travaux.

Enfin ma tendresse à Cathy et à mes enfants, qui supportent mes heures de travail irrégulières comme mes voyages occasionnels depuis toujours, et m' écoutent patiemment raconter des trucs incompréhensibles. Peut-être, qui sait, l'un d'eux se retrouvera-t'il un jour devant un manuscrit semblable, que je ne comprendrai pas mieux, et aura-t-il une pensée pour son père...

Table des matières

Chapitre 1

Introduction - français

1.1 Résumé

Ce mémoire marque une étape importante dans une carrière de chercheur longue déjà de 28 ans, depuis l'obtention de ma thèse en 1983 et mon arrivée la même année dans l'équipe Meije à l'INRIA Sophia-Antipolis jusqu'à mes travaux dans l'équipe Oasis depuis l'année 2000.

Les chapitres qui suivent concernent essentiellement ces dix dernières années d'activités, même si on pourra y trouver, au détour des pages, un certain nombre de références à mes travaux antérieurs. Je souhaite cependant, dans cette introduction, donner quelques éléments mettant en perspective les différents thèmes de ces années de recherche, et, peut-être, en souligner la cohérence et l'évolution.

Pendant ma thèse [T-83] [1], je me suis intéressé à la sémantique des langages de programmation, sous l'éclairage de leur sémantique axiomatique, dans le but d'établir une méthode pour prouver la correction de l'ensemble des composants d'un compilateur. Ce travail a donné lieu à une implantation prototype, utilisant le système de preuve de théorèmes LCF (Logic for Computable Functions [71, 76]).

Mon arrivée dans l'équipe Meije en 1983 (avec G. Berry et G. Boudol), à Sophia-Antipolis, a marqué un changement significatif de mes thèmes de recherches. Si le domaine général restait celui de la sémantique des langages et des programmes, avec un focus marqué sur les preuves de programmes, les méthodes changeaient tant sur les fondements théoriques que sur les techniques et les domaines d'application. Au plan théorique, mes travaux de thèse s'appuyaient sur des modèles sémantiques équationnels, et des techniques de réécriture. Ils portaient sur la preuve de programme, mais dans le cas très particulier de la preuve de correction de compilateurs, et à l'aide d'un logiciel de preuve de théorèmes interactif (LCF) [R-82,T-83,C-84]. A partir de 1983, mes travaux se sont orientés vers des sémantiques opérationnelles, et plus particulièrement comportementales, et j'ai participé au développement majeur, pendant les années 80 et 90, des travaux sur les algèbres de processus. En même temps, sur le plan des outils logiciels, j'ai participé au développement d'outils automatiques, de moteurs de vérification de modèles (model-checkers) fondés sur la théorie de la bisimulation, d'éditeurs pour des langages de spécification graphiques, mais aussi plus en amont, d'outils génériques pour l'étude des sémantiques comportementales.

[1] Les références bibliographiques de mes publications personnelles sont indiquées par catégorie de publication, avec le code suivant : E=Éditions, J=Journaux, C=Conférences internationales, W=Workshops, T=Thèses, R=Rapports, S=Standards et Logiciels. Les références générales, pour leur part, sont référencées par un simple numéro.

Dans la deuxième moitié des années 80, les recherches sur les algèbres de processus battaient leur plein, avec un foisonnement de travaux étendant les calculs (ou langages) originels CCS, CSP, ou ACP. Dans l'équipe Meije, nous étions particulièrement focalisés sur une famille de calculs asynchrones présentant des primitives de synchronisation très expressives : Meije-SCCS. Dans ce contexte, mais aussi dans les projets européens Concur et Concur2, j'ai participé à des recherches sur la forme des spécifications de sémantique opérationnelle, dans le but de caractériser syntaxiquement des calculs pour lesquels on savait donner des conditions suffisantes syntaxiques pour la terminaison des algorithmes de génération de modèle, donc utiliser des model-checkers. Ces travaux ont donné lieu à publication [C-90], mais aussi à la réalisation d'un outil nommé PAC (Process Algebra Compiler), dans le cadre d'une collaboration NSF-INRIA avec le Pr R. Cleaveland à N.C. State University (Raleigh, USA) [C-95].

Une application directe de ces résultats a permis la construction des outils de vérification AUTO [C-89], puis MAUTO [C-91b,C-92] et FC2Tools [24, 25], dans le cadre de la thèse de doctorat de Didier Vergamini [87]. AUTO était un vérificateur de propriétés et d'équivalences utilisant une représentation explicite des états, et des algorithmes de construction hiérarchique et de minimisation utilisant des bisimulations fortes ou observationnelles. Il était programmé en LeLisp. Les défis principaux, à l'issue de la thèse de Didier Vergamini, concernaient d'une part en amont, les passerelles entre différents langages de spécification ou de programmation et les formats d'entrée du système, d'autre part les performances des algorithmes de bisimulation. Nous avons répondu au premier point avec le système MAUTO, dans lequel les parties amont d'analyse syntaxique et de génération des modèles comportementaux étaient générés à partir des descriptions syntaxiques et sémantiques du langage (par le système ECRINS, précurseur du PAC), puis quelques années plus tard avec la conception du format intermédiaire FC2, format pivot de la plateforme de vérification de notre équipe. Le deuxième point, toujours en collaboration avec Didier Vergamini, a été implanté dans les outils FC2Tools, en C++, qui comportent des algorithmes de minimisation par bisimulation très efficaces, tant sur des représentations explicites qu'implicites (BDDs).

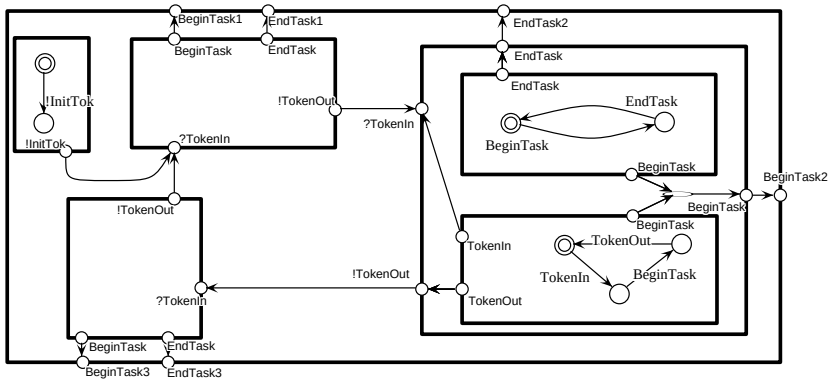


FIG. 1.1 – Dessin ATG de la spécification hiérarchique d'un Ordonanceur

Le compagnon indispensable de ces outils fut le système Autograph [80], conçu et implanté par Valérie Roy sous la direction de Robert de Simone, qui fournit aussi bien des éditeurs graphiques pour les automates comportementaux (systèmes de transition étiquetés) et les processus communicants, leur traduction en FC2, mais aussi la visualisation des résultats des outils de vérification, et en particulier des contre-exemples, lorsqu'une vérification échoue. Il est remarquable de noter que le système Autograph, dont le développement s'est arrêté en 1997, est encore

régulièrement téléchargé aujourd'hui et utilisé pour des besoins de recherche et d'enseignement. Le format FC2 a également été utilisé à cette époque par plusieurs systèmes de vérification de nos partenaires (CWB, CADP, UPAAL, Esterel, RTL, ...), et a ainsi permis des travaux de comparaison intéressants entre nos différentes techniques.

Un domaine d'application important de ces techniques était le langage de spécification LOTOS, pour lequel nous avons réalisé dans le cadre du projet Esprit Lotosphere une instanciation spécifique de MAUTO, destinée aux preuves de propriétés comportementales de programmes Basic Lotos [C-91A]. J'ai publié un article de synthèse présentant nos outils de vérification, et ceux de nos partenaires du projet CONCUR, dans la revue EATCS Bulletin [J-92]. La problématique d'application des outils de vérification en grandeur nature à de vrais langages de programmation restera une préoccupation et un défi important dans mes thèmes de recherche, et sous-tendra une partie importante de mes activités futures.

En 2000 j'ai rejoint Isabelle Attali et Denis Caromel au sein de la jeune équipe OASIS, toujours à l'INRIA Sophia-Antipolis. L'idée était de confronter les méthodes plutôt génériques que j'avais développées précédemment à un vrai langage de programmation, de définir les bases sémantiques permettant d'adapter nos méthodes génériques aux modèles de programmation créés par Oasis, de développer une plateforme logicielle intégrant des outils spécifiques de génération de modèles sémantiques avec des moteurs génériques de vérification, enfin de nous confronter à des cas d'étude réalistes dont la complexité dépassait de loin, nous le savions, les possibilités brutes des outils de l'époque.

Les défis étaient nombreux. Parmi les traits indispensables au traitement de langages et de cas d'étude réalistes, la prise en compte des types de données, d'appels de méthodes (mutuellement) récursives, de traitement des exceptions, étaient autant de problèmes critiques pour la définition de critères de finitude, ou de bonnes abstractions, de nos modèles. D'autres traits, plus spécifiques aux modèles de programmation Oasis, s'ajoutaient aux besoins de définition de modèles sémantiques : la prise en compte des appels asynchrones de méthodes entre objets actifs, avec leurs queues de requêtes et leurs futurs ; le support pour les futurs de première classe et leurs stratégies de mise à jour ; les communications de groupe ; les composants distribués enfin, formalisés dans le modèle GCM (Grid Component Model), avec leur structure d'encapsulation hiérarchique, leurs possibilités de reconfiguration dynamique, leurs contrôleurs non-fonctionnels [C-07b].

Dans un premier temps, dans le cadre des travaux de doctorat de Rabéa Boulifa [27], puis de Tomás Barros [14], nous avons défini un modèle sémantique comportemental étendant les modèles existants dans le monde des algèbres de processus, pour :

- prendre en compte la composition hiérarchique des processus, à un niveau sémantique très expressif : nous avons conçu pour cela une extension des vecteurs de synchronisation d'Arnold et Nivat (l'héritage de nos travaux sur le format FC2 est évident), permettant un codage sémantique flexible de modes de synchronisation très variés, plutôt que de nous limiter à un choix restreint d'opérateurs de parallélisme,
- incorporer un codage explicite des données, tant sous forme de communication "value-passing", que pour la description de topologies paramétrées de processus.

Ce modèle, baptisé pNets (pour *parameterized Networks of automata*), a été publié d'abord dans [C-04a] en 2004, puis dans ses formes les plus évoluées, dans [J-08,J-09]. Il nous a permis de définir des procédures de génération de modèles

comportementaux pour la plupart des “défis” listés ci-dessus, et en particulier : les objets actifs du modèle de programmation ASP [34] et de la bibliothèque ProActive, avec leurs queues de requêtes asynchrones et leurs futurs [C-03a,C-03b,C-04a], les composants distribués Proactive/GCM [C-05a,C-05b,C-06], les futurs de première classe [C-08a] et les communications de groupe [C-10].

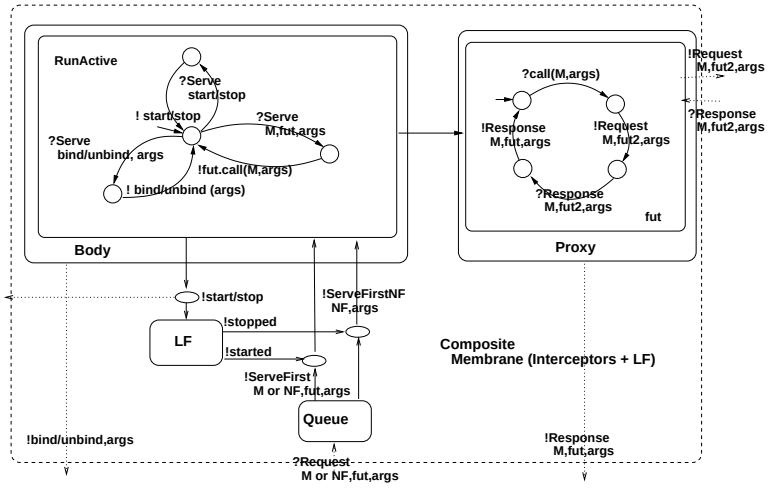


Fig. 1.2 – Structure pNets de la membrane d’un composant composite, FACS’05

En amont de la génération de modèles comportementaux, se pose le problème de l’abstraction : les programmes concrets sont trop complexes, trop détaillés pour être directement modélisés, il faut déterminer le juste niveau d’abstraction pour obtenir un modèle gérable (e.g. fini ou régulier), tout en capturant les propriétés que l’on veut garantir. Il y a essentiellement deux approches à ce problème : soit on utilise des mécanismes d’analyse statique de code source (éventuellement complétés par des annotations), soit on se base sur une modélisation préalable (langage de spécification ou formalismes à la UML), à partir de laquelle on générera du code “correct par construction”. Dans les deux cas on pourra aussi être amené à appliquer des techniques d’interprétation abstraite pour réduire encore la complexité du modèle sur lequel travaillent les moteurs de vérification.

Dans nos premiers travaux [C-03a,C-03b,C-04a], nous avons défini une méthodologie basée sur la première approche : extraction par analyse statique d’un graphe d’appel de méthodes et abstraction des domaines de données, aboutissant à la construction d’un modèle pNets codant la structure et la dynamique des objets actifs. Cette approche se heurte à de sérieux problèmes de précision de l’analyse statique d’une part, mais aussi de complexité, les techniques d’analyse se prêtant mal à des approches compositionnelles. Le passage aux composants distribués, par opposition aux objets actifs simples, permet de gagner de manière significative sur la précision de l’analyse, en particulier parce que la définition des composants impose de définir localement les besoins d’interactions avec le reste du système ou de l’environnement : ce sont les interfaces dites requises (ou client) du composant qui relaient les appels de méthodes distantes, et ces interfaces sont connues localement et statiquement. En même temps, nous avons exploré plusieurs voies pour la spécification du comportement et de l’architecture des systèmes de composants distribués, soit graphiques [C-07a,J-08], soit textuels [C-08c], dans le cadre de la thèse d’Antonio Cansado [32].

La mise en pratique de ces principes est une activité exigeante en terme de développement logiciel, surtout que notre ambition est de pouvoir mettre ces outils dans les mains de non-spécialistes, donc de mettre en place des interfaces utilisateur

de haut niveau, suffisamment intuitives pour cacher la complexité des modèles sous-jacents et des outils de vérification utilisés. Par ailleurs, notre vocation n'était pas de travailler sur les algorithmes et sur les moteurs de vérification eux-mêmes ; nous avons naturellement choisi des outils basés sur les équivalences de bisimulation, et utilisé des moteurs très performants, en particulier ceux de la plateforme CADP.

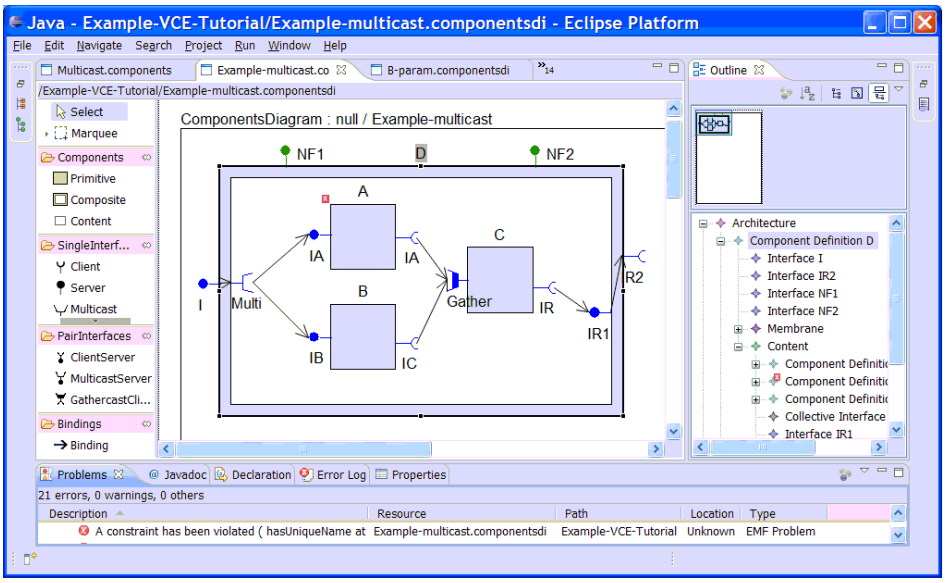


FIG. 1.3 – Editeur graphique de composants de la plateforme Vercors

Notre plateforme VerCors [W-06,C-05b,C-06,C-09] réunit notre éditeur graphique pour la spécification des architectures de composants, nos outils d'abstraction et de génération de modèles, et les passerelles vers les outils de minimisation et de model-checking de CADP. Le format pivot entre notre modèle pNets et les outils de CADP est le format Fiacre, que nous avons défini avec nos partenaires du projet (ACI Sécurité) Fiacre.

Bien sûr nous avons pu traiter, au fur et à mesure des progrès de ces travaux, un certain nombre de cas d'étude de complexité grandissante, depuis une modélisation du système de taxes électroniques chilien [C-04b] premier exemple de grande taille d'un modèle pNets ; du "Common Component Modeling Example" (CoCoME [J-08]), système de composants distribués hiérarchiques avec communication par messages synchrones ; au dernier en date [C-10] sur la vérification d'un protocole de vote avec queues de requêtes asynchrones et communication de groupe.

La prochaine étape, centrale en l'état actuel des travaux de l'équipe, consiste à prendre en compte les aspects dynamiques de nos systèmes. Le modèle GCM permet d'inclure dans les composants des contrôleurs gérant la plupart des aspects non-fonctionnels, y compris la reconfiguration (dynamique) des liaisons entre interfaces, le remplacement de composants, des protocoles résistants aux pannes, voire des stratégies complexes de type équilibrage de charge, optimisation de la consommation énergétique, ou adaptation à la demande. Les modèles comportementaux de ces applications seront de plusieurs ordres de grandeur plus complexes que ceux actuellement traités, et les propriétés à prouver dépendront de paramètres plus dynamiques.

Les pistes pour relever ces nouveaux défis passent vraisemblablement par des techniques de modélisation innovantes (nous avons par exemple mené quelques expérimentations dans le domaine des systèmes infinis), mais aussi par des interactions avec des techniques de preuve interactives, permettant de prouver des

propriétés génériques des modèles, et de réduire la complexité de la partie model-checking.

1.2 Structure du Document

Le chapitre 3 rassemble des éléments de comparaison avec d'autres travaux de recherche directement liés 'a notre sujet : systèmes distribués et modèles à base de composants ; sémantiques comportementales ; model-checking et plateformes de vérification.

Le chapitre 4 décrit notre développement du modèle sémantique pNets (parameterized networks of synchronized automata), et notre codage dans ce modèle de la sémantique du comportement des applications distribuées, en commençant par les objets actifs, et en intégrant progressivement toutes les fonctionnalités du modèle GCM. L'article principal sur le modèle pNets a été publié dans la revue *Annals of Telecommunications* [J-09] et est inclus ici.

Puis le chapitre 5 présente le développement de notre plate-forme de spécification et de vérification, depuis nos premiers développements de traducteurs de programmes ProActive vers les formats d'entrée des outils de vérification CADP, à travers les étapes successives de nos formalismes de spécification, jusqu'à la plate-forme actuelle VerCors et nos plus récents développements. Cette partie est illustrée par le document présenté au symposium FMCO'08 [C-09].

Le chapitre 6 décrit nos recherches en termes de formalismes de spécification, aussi bien graphiques (déjà inclus dans la plate-forme Vercors), que textuels, avec le langage de spécification JDC (Java Distributed Component). Ce dernier langage a été décrit dans un article à la conférence FACS'08 [C-08c], qui est inclus ici.

Le chapitre 7 est la dernière section technique de cette thèse, et présente trois importantes études de cas qui ont marqué notre travail. Dans chaque cas, nous soulignons les techniques utilisées pour modéliser le système, maîtriser la génération d'états, et prouver les propriétés requises. Ce chapitre est illustré par les publications décrivant le plus récent de ces cas d'utilisation [C-10, R-10].

Le chapitre 8 contient une analyse de l'état actuel de ce travail, et en expose les perspectives pour les années à venir.

Enfin, le chapitre 9 assemble les différents éléments de ma carrière en tant que chercheur, y compris mes activités de recherche et d'enseignement, principales collaborations, participations à des contrats de recherche, directions des étudiants (au doctorat, postdoc, mastère), et bibliographie.

Chapitre 2

Introduction - english

2.1 Summary

This dissertation marks an important step in a researcher career already 28 years long, since my PhD in 1983 and my arrival in the same year in the Meije team at INRIA Sophia-Antipolis, until my work in Oasis Team since 2000.

The following chapters mainly concern these last ten years of activity, even if we can find, along the pages, a number of references to my previous work. However I wish, in this introduction, to give some elements making perspectives on the different themes of these years of research, and perhaps, to highlight their consistency and their evolution.

During my thesis [T-83] [1], I was interested in the semantics of programming languages, from the point of view of their axiomatic semantics, in order to establish a method to prove the end-to-end correctness of a compiler. This work gives rise to a prototype implementation, using the theorem-prover LCF (Logic for Computable Functions [71, 76]).

My arrival in the team Meije in 1983 (with G. Berry and G. Boudol), in Sophia-Antipolis, marked a significant change in my research. If the overall domain was still the general semantics of languages and programs, with a focus on brand and program proofs, we changed both the theoretical grounds and on techniques and application domains. On the theory side, my thesis work was based on equational semantic models and rewriting techniques. My thesis focused on proofs of programs, but in the very particular case of the proof of correctness of compilers, and using a software interactive theorem proving [R-82,T-83,C-84]. In 1983, my work moved towards operational semantics, and more particularly behavioural semantics, and I contributed during years 80 and 90, to major developments on process algebras. At the same time, in terms of software tools, I participated in the development of automated tools, model verification engines (model-checkers) based on the theory of bisimulation, editors for graphical specification languages, but also of generic tools for the study of behavioural semantic.

In the second half of year 80, research on process algebras were in full swing, with an abundance of work extending the original process calculi (or languages) CCS, CSP or ACP. In the team Meije, we were particularly focused on a family of asynchronous calculi offering very expressive synchronization primitives : Meije-SCCS. In this context, but also in European projects Concur and Concur2, I par-

[1] The bibliographic references of my personal publications are referred by category, with the following code : E=Editions, J=Journals, C=Conferences (international), W=Workshops, T=Thesis (PhD), R=Reports, S=Standards and Software. The general references are referred by simple numbers.

ticipated in research on the form of operational semantics specifications, in order to characterize syntactically calculi for which we could give sufficient conditions for the termination of model generation algorithms, then use model-checkers. This work has was subject to publication [C-90], but also to the realization of a tool called PAC (Process Algebra Compiler), as part of an NSF-INRIA collaboration with Prof. R. Cleaveland at NC State University (Raleigh, USA) [C-95].

A direct application of these results allowed the construction of the verification tools AUTO [C-89], then Mauto [C-91b,C-92] and Fc2Tools, as part of the doctoral thesis of Didier Vergamini [87]. AUTO was a verifier for properties and equivalences using an explicit representation of states, and hierarchical construction and minimization algorithms using strong or observational bisimulation equivalences. He was programmed in LeLisp. The main challenges, at the end of the Didier's thesis, concerned first upstream bridges between different specification or programming languages and the input formats of the system, secondly the performance of bisimulation algorithms. We answered the first point with the Mauto system, wherein the front-end parsing and construction of behavioural models were generated from syntactic and semantic descriptions of languages (by the ECRINS software, precursor of the PAC), then a few years later with the design of the intermediate format FC2, central formalism of the verification platform of our team. The second point, again in collaboration with Didier Vergamini, was implanted in Fc2Tools tools, built in C++, which include very efficient bisimulation minimization algorithms, using both explicit and implicit (BDDs) representations.

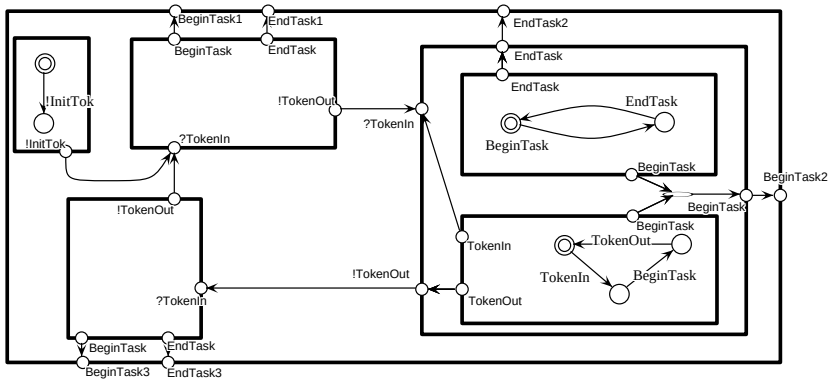


Fig. 2.1 – ATG drawing for a Scheduler hierarchical architecture

The companion of these tools was the Autograph system [80], designed and implemented by Valerie Roy under the direction of Robert de Simone, who provides both a graphical editor for behavioural automata (labeled transition systems) and communicating processes, a translator to the FC2 format, but also a visualization tool for the results of verification tools, in particular counter-examples, when the verification fails. It is remarkable to note that the Autograph system, whose development was stopped in 1997, is today still regularly downloaded and used for purposes of research and teaching. FC2 format has also been used at that time by several verification systems of our partners (CWB, CADP, UPAAL, Esterel, RTL, ...), and thus allowed an interesting comparison work between our different techniques.

An important application area of these techniques was the specification language LOTOS, for which we implemented within the Esprit project Lotosphere a specific instantiation of Mauto, intended for proofs of behavioural properties of Ba-

sic Lotos programs [C-91A]. I published a review paper presenting our verification tools, and those of our partners in the project CONCUR in the journal EATCS Bulletin [J-92]. The problem of using verification tools for real-size applications, and real programming languages, will remain a major concern and challenge in my research topics, and will underpin a significant part of my future activities.

In 2000, I joined the young team of Isabelle Attali, also at INRIA Sophia-Antipolis. The idea was to check the applicability of the generic methods I had previously developed to a real programming language, to define the semantic grounds for adapting our generic methods to the programming models created by Oasis, to develop a software platform integrating specific tools for generation of semantic models with generic verification engines, finally, to address case studies of realistic complexity which far exceeded the crude possibilities of state-of-the-art tools of that time.

The challenges were numerous. Among the features essential for realistic language and case-study processing, taking into account the data types, (mutually) recursive method calls, exceptions handling, were all critical problems for the definition of finiteness criteria, and of good abstractions, of our models. Other features, more specific to Oasis programming models, also required a precise definition of their behavioural semantics, in terms of pNets generation : management of asynchronous methods calls between active objects, with their future and their request queues, support for first class futures and their update strategies, group communications ; and finally distributed components as formalized in the Grid Component Model (GCM), with their hierarchical structure, encapsulation, their possibilities for dynamic reconfiguration, their non-functional controllers [C-07b].

Initially, as part of doctoral work of Rabea Boulifa [27] and Tomas Barros [14], we defined a behavioural semantic model extending the existing models in the world of process algebras, to take into account :

- hierarchical composition of processes, hat a very expressive semantic level : we developed to this aim an extension of synchronization vector technique of Arnold and Nivat (the heritage from our work on the FC2 format is obvious), allowing a flexible semantic coding of various synchronization modes, rather than limiting ourselves to a limited choice of parallel operators,
- explicit management of data, both for "value-passing communication", and for the description of parameterized process topologies.

This model, called pNets (for *Parameterized Networks of Automata*), was first published in [C-04a] in 2004, and later in its most developed form, in [J-08,J-09]. It allowed us to define procedures for generation of behavioural models for most of the "challenges"lists above, and specifically : active objects of ProActive programming model, with their asynchronous request queues and their futures [C-03a,C-03b,C-04a], distributed components ProActive/GCM [C-05a,C-05b,C-06], first class futures [C-08a], and group communications [C-10].

Ahead of the generation of behavioural models, is the problem of abstraction : concrete programs are too complex, too detailed to be modeled directly, and we must determine the right level of abstraction to obtain a manageable model (ed finite or regular), while capturing the properties we want to guarantee. There are basically two approaches to this problem : either we use static analysis methods on the source code (possibly supplemented by annotations), or we start with an early modeling phase (using a specification language or a modeling formalism a la UML), from which we generate code "correct by construction". In both cases we may also have to apply abstract interpretation techniques to further reduce the complexity of the model on which the verification engines will work.

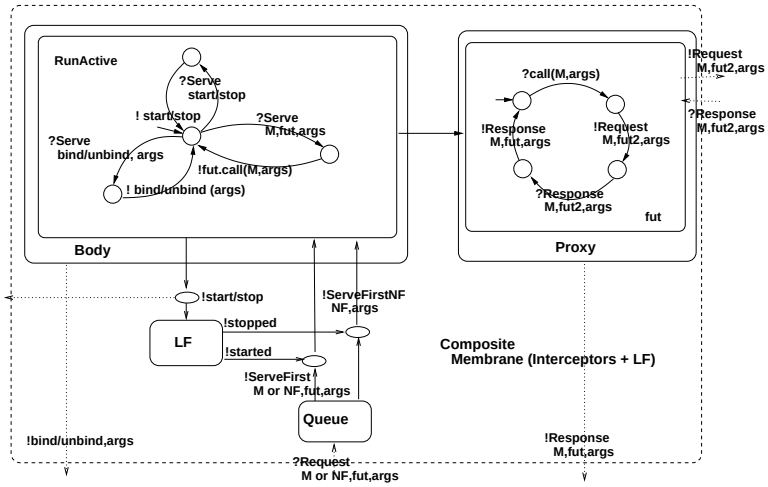


FIG. 2.2 – pNets structure of a composite component membrane, FACS'05

In our early work [C-03a C-03b, C-04a], we defined a methodology based on the first approach : extraction by static analysis of a method call graph and abstraction of data domains, resulting in the construction of a pNets model coding the structure and dynamics of active objects. This approach faces serious problems of accuracy of the static analysis on the one hand, but also of complexity, because analysis techniques are ill-suited for compositional approaches. The move to distributed components, as opposed to simple active objects, saving significantly on the accuracy of the analysis, especially because the definition of components requires a precise definition of interactions with the rest of the system or the environment (through required interfaces). At the same time, we explored several avenues for the specification of the behaviour and architecture of distributed component systems, either graphical [C-07a, J-08] or textual [C-08c], in the context of Antonio Cansado PhD thesis [32].

The implementation of these principles is a demanding activity in terms of software development, especially since our ambition is to put these tools in the hands of non-specialists, thus establishing high-level user interfaces, intuitive enough to hide the complexity of the underlying models and verification tools. Besides, our aim was not to work on algorithms and verification engines themselves ; we use existing state-of-the-art engines from the area of bisimulation-based verification, in particular those from the CADP toolset.

Our Vercors platform [W-06,C-05b,C-06,C-09] associates our graphical editor for the specification of component architectures, our tools for abstraction and model generation, and bridges to the minimization and model checking tools of CADP. The pivot format between our pNets model and the CADP tools is the Fiacre language, that we defined with our partners in the Fiacre project (ACI Safety).

Of course we have handled, along the progress of these work, a number of case studies of increasing complexity, a modeling of the emerging Chilean electronic tax system [C-04b], which was a first example of large a pNets model ; the "Common Component Modeling Example"(CoCoME [J-08]), hierarchical distributed component system with synchronous message passing ; and the latest to-date [C-10] a voting protocol with asynchronous requests queues and group communication.

The next step, essential in the current state of our team work, is to take into account the dynamic aspects of our systems. The GCM can include component controllers managing most non-functional aspects, including the (dynamic) reconfiguration of bindings between interfaces, replacements of components, failure re-

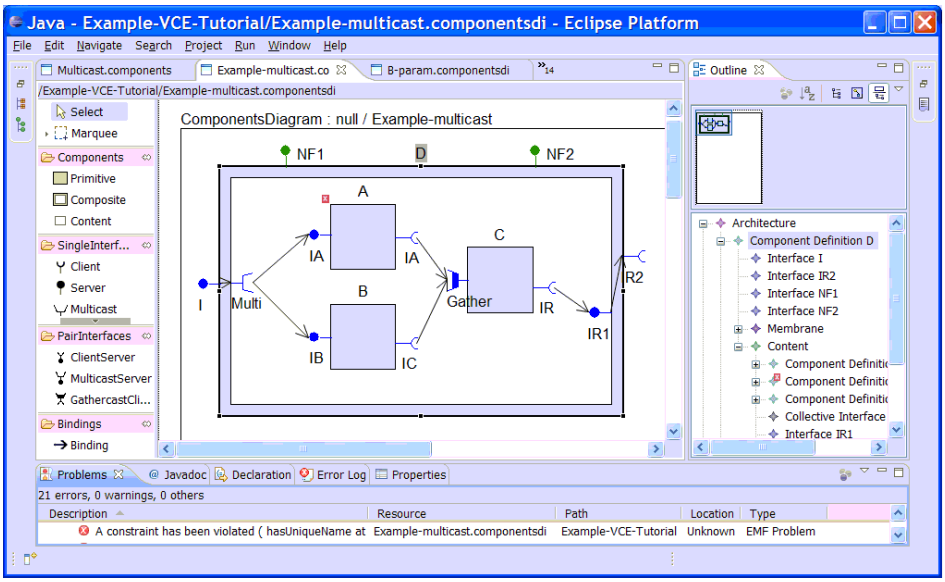


FIG. 2.3 – Vercors Graphical Component Editor

sistant protocols, or even complex strategies like load balancing, optimizing the energy consumption or adaptation to external demands. Behavioural models for these applications will be several orders of magnitude more complex than those currently treated, and properties to prove will depend on more dynamic parameters.

Tracks to address these new challenges are likely to pass through innovative semantic representation techniques (for example, we conducted some experiments in the field of infinite systems), but also by interactions with interactive proof techniques, for proving the properties of generic models, and reduce the complexity of the model-checking activity itself.

2.2 Document Structure

Chapter 3 gathers elements of comparison with related work : Distributed systems and Component Models, Behavioural Models and Abstract Models, Model-checking and Verification Platforms.

Chapter 4 will describe our development of a semantic model pNets (parameterized networks of synchronized automata), and our encoding in this model of the behavioural semantics of distributed applications, starting with active objects, and incorporating progressively all features of the GCM model. The main article on the pNets model was published in the *Annals of Telecommunication* journal [J-09], and is included here.

Then, chapter 5 presents the development of our specification and verification platform, from the early development of translators from ProActive programs to the input formats of the CADP verification toolset, through the successive steps of our specification formalisms, to the current VerCors platform and our most recent developments. This part is illustrated by the paper presented at the FMCO'08 symposium [C-09].

Chapter 6 describes our research in terms of specification formalisms, both graphical (as included in the VerCors platform), and textual, with the Java Distributed Component specification language (JDC). This last language was described a paper at the FACS'08 conference [C-08c], which is included here.

Chapter 7 is the last technical section of this thesis, and presents three important case-studies that have marked our work. In each case, we emphasize the techniques used for modeling the application, mastering the state-generation process, and proving the required properties. The chapter is illustrated by the publications describing the most recent of these use-cases [C-10,R-10].

Chapter 8 contains an analysis of the current state of this work, and exposes perspectives for the forthcoming years.

Finally, chapter 9 assembles the various elements of my career as a researcher, including my research and teaching activities, main collaborations, involvement in research contracts, students direction (at PhD, postdoc, master levels), and bibliography.

Chapitre 3

Related Work

In this chapter we give an overview of research related to our subject, commenting the relations and differences when possible. We start with (distributed) component models, then describe various (low-level) formalisms used to model distributed systems, and list a number of existing verification tool-sets, dedicated to various programming languages, or based on different model-checking methods. Finally we mention shortly a number of more recent approaches, that are not yet fully integrated in available platforms, but that are worth considering for experimentation and further work.

Distributed systems and Component Models

Our focus in this work is not the design of (software) components models, but our starting point was necessarily the choice of a programming model for distributed applications on which we could build our semantic definitions and our verification methodology. So we start our related work section by a short panorama of models for distributed programming and distributed component systems.

In this section, we review some software component models that are targeted at the programming of distributed applications, and development of middleware, taking into account constraints raised by distribution. We start with some of the most commonly known models for a component-oriented approach [75] to distributed computing : the Common Component Architecture (CCA), the CORBA Component Model (CCM), and the Service Component Architecture (SCA) after which we will describe the main features of Fractal and GCM, and motivate their choice for our work.

CCA has been defined by a group of researchers from laboratories and academic institutions committed to specifying standard component architectures for (Grid) high performance computing [10, 35]. In CCA a component "is a software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities. A component has a clearly defined interface and conforms to a prescribed behaviour common to all components within an architecture." Currently the CCA Forum (www.cca-forum.org) gathers documents, projects and other CCA-related work including the definition of a CCA-specific format of component interfaces (Babel/SIDL - SRPC Interface Description Language) and framework implementations (Ccaffeine, Xcat).

However, the CCA model is non-hierarchical, thereby making it difficult to handle the distributed and possibly large set of components forming a Grid or Cloud application in a structured way. Indeed, hierarchical organization of a compound application can prove very useful in getting scalable solutions for management

operations pertaining to monitoring, life-cycle, reconfiguration, physical mapping on infrastructure resources, load-balancing, etc. Unfortunately, the CCA model is rather poor with regards to managing components at runtime. It means a CCA component per se does not have to expose standard interfaces dedicated to non-functional aspects as it is the case for Fractal components. This makes it hard to realize certain features, for instance, dynamic reconfiguration based on observed performance or failures, or elastic adaptation to "business" requirements. However, some implementations of the model, like e.g. XCAT, can provide some additional components (like an Application Manager) dedicated to manage the non-functional aspects of a CCA-based application.

CCM is a component model defined by the Object Management Group (OMG). The CCM specifications include a Component Implementation Definition Language (CIDL) ; the semantics of the CORBA Component Model (CCM) ; a Component Implementation Framework (CIF), which defines the programming model for constructing component implementations and a container programming model. Important work has been performed to turn the CCM in a Grid component model, like GridCCM [44].

In CCM, the ADL is able to deal with distributed resources but it is outside the scope of the specifications to describe how such a description has been generated. However, this task requires a high level of knowledge of the application structure as well as the resource properties. This approach is not satisfactory for Grids or Clouds where resources are provided dynamically.

Even if CCA and CCM components can fit into a distributed infrastructure, they are not designed as being per se distributed. Consequently, it is quite unnatural to use them to build parallel entities to be mapped onto a set of distributed resources, or having the capability to self-adapt to the changing context. By contrast, the Enterprise Grid Alliance effort [85] is an attempt to derive a common model adopting Grid technologies for enhancing the enterprise and business applications. The model, which is aligned with industry-strength requirements, strongly relies on component technology along with necessary associations with component-specific attributes, dependencies, constraints, service-level agreements, service-level objectives and configuration information. One of the key features that the EGA reference model suggests is the life-cycle management of components which could be governed by policies and other management aspects. The level of this specification, however, is very coarse-grain focusing on system integration support rather than providing an abstract model and specification for Grid programming which is the main goal of GCM.

University of Kansas has developed the verification environment Cadena [56, 36], and its meta-modeling language CALM [61], for the specification, verification, and development of CCM applications.

SCA is a set of specifications proposed by the OASIS standard body [4], which describe a hierarchical model for building applications and systems using a Service-Oriented Architecture (SOA). The SCA specifications were first published in Nov. 2005, including the Assembly Specification, the Client and Implementation Specification for Java and the Client and Implementation Specification for C++. Implementations include open-source tools developed by the Tuscany project [1] of the Apache Software Foundation, or the SCOrWare french project [2].

Fractal and GCM Fractal [29] is a general component model which is intended to implement, deploy and manage (i.e. monitor, control and dynamically configure) complex software systems, including in particular operating systems and middleware. The Grid Component Model (GCM) [11, 15] is a Fractal extension providing specific features for programming distributed systems, typically on Grid, P2P, or Cloud infrastructures.

Among Fractal's peculiar features, below are those that motivated us to select it as the basis for the GCM.

- Hierarchy (composite components can contain sub-components), to have a uniform view of applications at various levels of abstraction.
- Introspection capabilities, to monitor and control the execution of a running system.
- Reconfiguration capabilities, to dynamically configure a system.

To allow programmers to tune the control of reflective features of components to the requirements of their applications, Fractal is defined as an extensible system. In addition, the Fractal specification is a multi-level specification, where depending on the level some of the specified features are optional. That means that the model allows for a continuum of reflective features or levels of control, ranging from no control (black-boxes, standard objects) to full-fledged introspection and intercession capabilities (including e.g. access and manipulation of component contents, control over components life-cycle and behaviour, etc). Fractal does not constrain the way(s) the GCM can be implemented, but it provides a basis for its formal specification, allowing us to focus only on the Grid-specific features. Eventually, platforms implementing the GCM should constitute suitable grid programming and execution environments. ProActive offers one such implementation [10].

Behavioural Models, Abstract Models

Historically, models of behaviours were defined in terms of semantic-level representations, ranging from core Labelled Transition Systems (LTS), from the very beginning of the process algebra era (see [72, 19]), and the synchronization vectors of Arnold and Nivat [8], to Milner's π-calculus [73]. LTS is, without contest, the most often used model for the representing behaviours in analysis and verification toolsets. At the other end of the spectrum, the π-calculus has only been used in a few research prototypes, because its high expressiveness comes with a very high complexity of the related representations and algorithms.

Most established approaches, on the other side, are using intermediate formats with data, that can be unfolded to finite-state structures. This is the case e.g. for the CADP toolbox [47], or for the SPIN model-checker [58]. These systems have a usual input language (Lotos and Promela, respectively) that can be used directly to model the systems to be analyzed. But many users are implementing translators from their source language into Lotos or Promela. Both of these modeling languages have very expressive constructs, high-level process definition and communication mechanisms (with gate negotiation in Lotos, or more classical channels in Promela), and typed data manipulation. Naturally, Lotos and Promela programs have infinite behaviours, due to unbounded data on one hand, and to recursive process definition on the other hand. The state exploration engines of the model-checkers have mechanisms to specify bound domains for data, or bound expansion of the behaviours.

When model-checking software systems written in usual programming languages, there is an inescapable step dedicated to build an abstract, smaller and

more manageable, version of the original program.

> Quoting Patrick Cousot : Model-checking exhaustively verifies temporal properties on a finite model of hardware or software computer systems [38]. This abstraction of a system into a model is often left implicit. Abstract model checking, as formalized by abstract interpretation, makes this abstraction explicit [39], [42]. Model-checking is reputed to be terminating, sound, and complete on the model. From an abstract interpretation point of view, relating the system to its model, it may be sound on the model but unsound on the system (e.g. the model is correct for safety properties but wrong for liveness properties), it is often incomplete (no finite model can cover the specified behaviors of the system [78]) and, in practice, may explode combinatorially. In all cases abstract interpretations of the system into a model have to be considered.

There is one early work from Cleaveland & Riely [40], that defines a convenient mechanism for building abstract (finite) behavioural models of distributed applications, starting from the specification of abstract interpretations of the data domains, while respecting safety and liveness properties of the original concrete (infinite) programs.

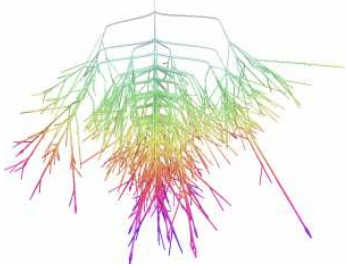
Finding the right compromise between very abstract (and small) models, and more precise (but larger) models may be a difficult task. Part of the solution relies on program slicing (keeping only the segments of the program that influences the property or the set of properties to be proved) (see e.g. [21, 45]). Some authors have proposed iterative methods, where failures of the model-checking engine are analyzed and are used to refine the abstraction, until the model is precise enough to prove the goals [37, 55, 54].

Model-checking and Verification Platforms

Many works have been done based on process algebra foundations, starting in the eighties with research tools implementing the CCS, CSP, or ACP algebras [J-92]. Some of these tools have given birth to systems with a more developer-oriented specification language :

The FDR2 tool [28] offers a high-level language for expressing CSP models, and an internal machine-readable dialect of CSP [84] using a specific expression language, more adapted to generate the models needed by the verification engines. Strictly speaking, FDR2 is a *refinement checker* rather than a model-checker, in the sense that it compares two LTSs,the system and its specification, with respect to some specified refinement relation (trace, failures or failure/divergence).

The μCRL tool, and its successor mCRL2 [52, 51], based on the ACP process algebra, are offering an expressive language with data manipulation, and rich analysis tools for linearization, simulation, state-space exploration and generation and tools to optimize and analyze specifications. Moreover, state spaces can be manipulated, visualized and analyzed.



A colorful transition system visualization with the mCRL2 toolset

The UPAAL system [18, 63, 16] was born also in the process algebra era, and implements a number of extensions of the original "pure" calculi, including data, timed and probabilistic calculi. It is still one of the extensively used system in research and education, and also offers a commercial version.

The CADP toolset [48] is one of the proeminent platforms for the specification, verification, and testing of distributed systems in the European academic landscape. Initially built as an environment dedicated to LOTOS programs, it was progressively open to handle a number of different input formalisms, through several input formats, together with an extensible API. The toolset includes engines for building the state-space of systems in a hierarchical way, building and manipulating LTSs on distributed infrastructures, minimizing LTSs along several behavioural equivalences, model-checking properties, checking equivalences between systems, building test suites, evaluating performances, etc.

In parallel with all these "process algebra" based tools, most of them from European research labs, there were a very important family of tools born on the other side of the Atlantic, mostly based on trace semantics and linear time logics, rather than bisimulation semantics and branching time logics. The most renowned of these are certainly SPIN [59, 58] (linear time), SMV [30] (branching time, state-based) and its real-time/hybrid extensions [31, 64], or SLAM [13, 12] (linear-time, with focus on C-code abstraction). These tools have been extensively used for hardware and embedded system verification, but they also have been used (by translation to their respective input formalisms), in the area of protocols and distributed systems.

There are also a small number of researches dedicated more specifically to component oriented verification. This is the case e.g. of SOFA, STSlib, and Kmelia... :

The SOFA system [77, 3, 60] is dedicated to the development of large, distributed software systems, based on hierarchical components. It uses a model of "behavior protocols" for the specification of possible interactions between components, and notions of compatibility for safe component assembly, and of hierarchical refinement.

The STSLib library [46] provides a formal component framework that synthesizes components from symbolic protocols in terms of Symbolic Transition Systems (STS). Just as pNets, STS concisely represents infinite systems, however, STS rely on Abstract Data Types (ADT) which are more expressive than the Simple Types used in pNets but less intuitive for software engineers. Both formalisms rely on (N-ary) synchronization vectors, but in STS they are static whereas in pNets they are dynamic. STSLib synthesizes components based on their STS protocols ; a controller interprets the STS protocol and data from the ADT is implemented (and generated) in Java. The communication in STS components is rather low-level ; both emitter and receiver must agree exchange a message, although there is no explicit notion of required nor provided services.

Kmelia [9, 6, 5] is a component specification model based on the description of complex services. Kmelia and its toolbox COSTO can be used to model software architectures and their properties, these models being later refined to execution platforms. It can also be used as a common model for studying component or service model properties (abstraction, interoperability, composability), using various verifications toolsets, including CADP, MEC5, and Atelier-B.

Chapitre 4

Behavioural Models

4.1 Summary

The paper included in this chapter has been published in January 2009 in the Annals of Telecommunications journal, issue entitled "Component-based architecture : the Fractal initiative". This article is the outcome of a series of workshop and conference papers presenting our efforts for building a semantic model dedicated to the analysis of distributed software.

In the Introduction chapter we have motivated the need for such a *behavioural model*, that would be :

- flexible enough to address a large set of distributed programming concepts,
- compact enough to be the basis for a manageable intermediate format,
- and defined with the idea of opening the possibilities for convenient "abstractions" towards specific classes of decidable models (finite, regular, etc.).

This story started in years 2002-2003. The original question was :

> ***Can we use existing formalisms and existing semantic models to lift verification methods from "academic" calculi (process algebras and their natural LTS-based behavioural semantics), to real languages, to support the analysis of Java/ProActive applications ?***

The challenge was to address the kernel features and paradigms of ProActive, including Java object/method basic structure, asynchronous communication using ASP/ProActive remote procedure calls, programmable request selection policies, and a small subset of data and data-types manipulation, sufficient to build realistic small examples. This was quite different from the work we developed previously in the Meije team : instead of defining a kernel formalism (a calculus or an algebra) as small and expressive as possible, dedicated to our specific interest of the moment, we had to tackle existing languages, with their large set of constructs,

Among the formalisms mentioned in the state of the art (Section 3), Lotos was certainly the closest to our goals, inherited from the process algebra decades, it had good capabilities for process definition, parallelism construction, and communication expression. I had worked a lot with Lotos, in particular in the context of the Lotosphere Esprit project, and developed specific behaviour model construction tools, and successful verification case-studies, for a sub-language without data called Basic Lotos. There were at least three important reasons why Lotos was not a good candidate to define the behavioural models of ProActive : its (algebraic) data type descriptions are very far from object oriented types of Java, so the encoding would have been complex ; the parallel constructs of Lotos, with their original

"offer negotiation" are very different from the more pragmatic remote method call of ProActive, and totally unable to encode the multicast communication of our programming models ; and, last but not least, the recursive definition of processes in Lotos is too powerful for deciding conveniently of the finiteness of process architectures. So it failed to meet most of the items listed at the beginning of this chapter.

In the context of Rabéa Boulifa's PhD doctoral thesis (2002-2004), we explored the idea of building by code analysis an "extended method call graph" (XMCG), encoding : 1) the dynamic structure of (possibly recursive) method calls, 2) the special structure for managing remote method calls, with their future proxies and their request queues, 3) an abstraction of data-types and data-variables. From these XMCGs, a behavioural semantics defined by Structural Operational Semantic (SOS) rules was able to produce pNets semantic models.

The pNets model is a tree-like structure which leaves are Labelled Transition Systems with data values (similar to the "Symbolic Transition Systems" of Lin and Ingolfsdir [65]), and nodes (Networks) have the role of synchronizing the communication events of subsystems (similar to the "Synchronization Vectors" of Arnold and Nivat [8]). See the formal definition in [J-09, pages 30-31]. The main originality of this model is the idea that Networks themselves are parameterized, i.e. indexed by data-variables, that allows us to model directly the parameterized architectures (pipelines, rings, vectors or matrices of processes, etc.), and also dynamic message routing or dynamic architecture configuration. This was first published in workshops at [C-03a], [C-03b], [W-04], and at the FORTE conference in September 2004 [C-04a], where the denomination ***pNets*** first appeared.

In the Forte'04 version, we used pNets solely for encoding the behaviour of Java method calls (potentially recursive), and ProActive mechanisms for active objects, including asynchronous communication, management of request queues, and of future values. This encoding uses heavily the parameterized Network structure of the pNets, where e.g. the set of invocations of a given method in an active object is encoded by an unbounded family of processes in the pNets, indexed by a natural number representing the successive calls of the method [C-03b]. In the same paper, we described the method for static analysis of Java/ProActive code, building the extended method call graph, and for building safe abstraction of data domains. In Rabéa Boulifa PhD thesis [27], we proved that this construction terminates. But there were strong open questions at this point, the most important being the *imprecision of static analysis*. In particular, consider the case of a piece of code containing a loop structure over an indexed structure (pipe-line, ring, vector, matrice, etc...) containing active objects ; a statement addressing these objects by an indexed reference in the structure will refer both to the local and to remote objects, so it is impossible to determine statically which calls correspond to the local object, and which calls correspond to remote objects. This fact has serious consequences in terms of the kind of properties that the model can prove, for such parameterized topologies.

The next step was to consider distributed components, as a mean to answer this problem. One of the main feature of components is that dependencies upon external (= remote) components are explicit, through the use of required interfaces. A number of component models also feature hierarchical structures, that fit naturally well with our semantic models (hierarchical networks + bisimulation). We considered Fractal components [29], as a flexible and extensible model offering the desired properties, and also the Grid Component Model [15] (GCM), that is the distributed extension of Fractal compatible with ProActive active objects. In [C-05a,C-05b],

we published the description of pNets model generation for 1) hierarchical components, 2) Fractal hierarchical components with life-cycle and binding controllers, 3) GCM components, i.e. Fractal components encapsulating active objects at both primitive and composite levels.

Naturally, it took us a couple years before publishing a journal version summarizing this work [J-09]. The contributions of this article are :

- a formal definition of the pNets model, together with its instantiation operator, and a product operator defined on instantiated nets,
- Four different application cases of increasing complexity, defining the model generation for ProActive active objects, hierarchical components, Fractal components, GCM components.

The version described in the Annals paper did not cover some advanced features of GCM that are very useful for our distributed application area, in particular first class futures and group communication, that we developed in the same period than the journal paper, and presented at FACS [C-08a] and at FMCO'2008 [C-09].

First class futures allow components to pass references to futures within requests arguments. In the ProActive library, futures are managed by proxy mechanisms, which act as relays between the request caller and the remote service. This is an important feature for GCM applications, permitting more asynchrony between components. But it also raises new questions for verification, because of dependencies introduced between the futures passing and update occurrences. There are a number of strategies for future updates, that require different kinds of future handling mechanisms in the implementation, and naturally different kind of future proxies in our models. We have published some of these variants in [C-08a], and have described the basis of this proxy model in [C-09, page17].

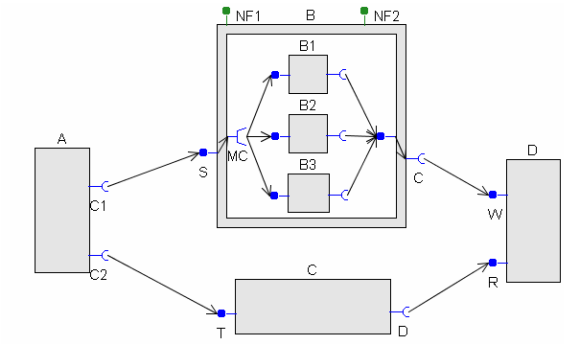


Fig. 4.1 – Example of Multicast and Gathercast communication

Collective interfaces, in the GCM model, introduce a new structuring feature in component architectures : *Multicast interfaces* are managing 1 to N communications, delivering their messages in a broadcast manner, and collecting results asynchronously. *Gathercast interfaces* is the dual mechanism, gathering requests along a number of connected "client" components, assembling them in a single request transmitted to the service interface, then distributing the result of the computation along some specific policy (that could be duplicating, scattering, or others). Here again we model this mechanism using *Group proxies*, that encode the policies and the API methods of both multicast and gathercast interfaces. The synchronization variants make use of the flexibility of pNets parameterized synchronization vectors to encode directly point to point (interleaved) or multi-point (multicast) communication. We shall give more feedback on this part in the Use-cases chapter 7.1.

Evaluation Our pNets model is a low-level semantic model similar in spirit to label transition systems, and not a process algebra or a calculus defining specific operators and concurrency structures. Still it includes a powerful mechanism for encoding various kinds of parallel composition and inter-process communication ; this hierarchical structure has two essential benefits : 1) it is close to program structure, easier to generate, and producing smaller models, than bare transition systems ; 2) it leads naturally to bisimulation-based semantics, and hierarchical approaches to verification, using compositional minimization. Moreover it encodes explicitly data types and variables, again producing smaller encodings, and opening the way to different types of verification methods (infinite systems, logics with counters, etc...).

One could argue that Lotos[23] or μ-CRL [53, 52], for example, twenty years ago, were already attempts to build such highly structured formalisms with rich data manipulation. And indeed full verification methodologies and toolsets were built for these languages [47, 51]. But our goals are quite different : Lotos and μ-CRL are *languages* with a specific set of operators, and a fixed communication and concurrency semantics, whereas pNets, having a lower-level synchronization mechanism, support a large variety of parallel operators and communication mechanisms (we shall use this flexibility when modeling group communication in section 7.1). We choose not to include any mechanism for recursivity (as in Lotos) nor high-order process construction as in the *pi*-calculus [73] : this would leave more difficulties in later steps of our methodology, when translating pNets models to the various input formalisms of model-checking and verification engines.

So the final compromise is a model flexible enough and expressive enough to encode a variety of programming structures, parallel constructions, and communication modes. It is also restricted in such a way that translation to finite/regular models remains simple.

At this point, we have set up the basic framework defining our semantic model, but we are still quite far from being able to apply it to the analysis of practical applications. In particular, experimentation was mandatory, for proving that the approach was indeed useful. Even for small experiments, some significant tooling was unavoidable. This will be the object of the next chapter.

4.2 Paper from *Annals of Telecommunications, Jan. 2009*

Ann. Telecommun. (2009) 64:25–43
DOI 10.1007/s12243-008-0069-7

Behavioural models for distributed Fractal components

Tomás Barros · Rabéa Ameur-Boulifa · Antonio Cansado · Ludovic Henrio · Eric Madelaine

Received: 30 July 2007 / Accepted: 16 July 2008 / Published online: 10 January 2009

Abstract This paper presents a formal behavioural specification framework for specifying and verifying the correct behaviour of distributed Fractal components. The first contribution is a parameterised and hierarchical behavioural model called *pNets* that serves as a low-level semantic framework for expressing the behaviour of various classes of distributed languages and as a common internal format for our tools. Then, we use this model to define the generation of behavioural models for applications ranging from sequential Fractal components, to distributed objects, and finally to distributed components. Our models are able to characterise both functional and non-functional behaviours and the interaction between the two concerns. Finally, this work has resulted in the development of tools allowing the non-expert programmer to specify the behaviour of his components and (semi)automatically verify properties of his application.

T. Barros
Universidad de Chili, Ejército 441, Santiago, Chile
e-mail: tomas.barros@niclabs.cl

R. Ameur-Boulifa
GET/ENST/LabSoC, Telecom Paristech, BP 193,
06904 Sophia-Antipolis Cedex, France
e-mail: Rabea.Ameur-Boulifa@telecom-paristech.fr

A. Cansado · L. Henrio · E. Madelaine (✉)
INRIA Sophia-Antipolis, CNRS, UNSA,
INRIA, Oasis. 2004, Route des Lucioles, BP 93,
06902 Sophia-Antipolis Cedex, France
e-mail: Eric.Madelaine@sophia.inria.fr

A. Cansado
e-mail: Antonio.Cansado@sophia.inria.fr

L. Henrio
e-mail: Ludovic.Henrio@sophia.inria.fr

Keywords Hierarchical components · Distributed asynchronous components · Formal verification · Behavioural specification · Model-Checking

1 Introduction

Component models provide a structured programming paradigm allowing a better reusability of programs by the fact that both provided/required services and application structure are expressed statically in the composition. This takes even more importance as the structure of distributed components acts as an abstraction for the component distribution. However, this architectural description is not always sufficient. Indeed, in order to be able to safely compose "off-the-shelf" or even dynamically discovered components, a form of specification language is required. Such a specification can only rely on the existence of some well defined semantics for the underlying programming language or middleware.

Among the existing component models, *Fractal* [1] provides the following crucial features: the explicit definition of provided/required interfaces for expressing dependencies between components; a hierarchical structure allowing to build components by composition of smaller components and the definition of non-functional features through specific interfaces, providing a clear separation of concerns between functional and non-functional aspects.

Globally, our work is placed in the context of large-scale distributed applications. This work is strongly related to programming models that aim at easing the programming of distributed applications by providing

high-level abstractions of distributed features together with an efficient implementation of these features. More precisely, we rely on the *Grid Component Model (GCM)* [2], which extends Fractal by addressing large-scale distributed aspects of components.

Moreover, in a distributed context, adaptive components are necessary in order to adapt the application to constantly evolving environments and evolving requirements in terms of quality of services. Our work is intended to be adapted to the verification of autonomous systems adapting and reconfiguring themselves in order to better match dynamic requirements of the application.

Our main objective is to provide tools to the programmer of distributed components in order to verify the correct behaviour of his program. We require those tools to be intuitive and user-friendly for them to be usable by non-experts of formal methods. To this end, we build an analysis toolset, including state-of-the-art model-checking tools; at the heart of this platform lie the model generation tools that are the subject of this article. In this context, the choice of the behavioural model is crucial: it has to be compact, expressive enough represent the behavioural semantics, but not too much, to allow an easy mapping to the model-checker input format.

Related work Historically, models of behaviours were defined in terms of semantic-level calculi, ranging from core Labelled Transition Systems (LTS), from the very beginning of the process algebra era (see [3, 4]), and the synchronisation vectors of [5], to Milner's π-calculus [6]. LTS is also, without contest, the most often used model for the representing behaviours in analysis and verification toolsets. At the other end of the spectrum, the π-calculus has only been used in a few research prototypes, because its high expressivity comes with a very high complexity of most related algorithms.

Early verification tools were using internal formats with a very simple structure, featuring no data parameters; even intermediate formats used to interface different tools were kept at a very low level. However, introducing data in those languages appeared quickly as being very beneficial both for compactness and for expressiveness.

For example, in the CADP toolbox [7], the internal model is a version of Petri nets with data that can be later unfolded (eventually on-the-fly) into LTSs suitable for model-checking. Recently, a new semantic-level format named NTIF [8], resembling our pLTS, has been devised as a more structured and compact intermediate form between LOTOS or ELOTOS programs and the CADP engines.

In a similar way, the SPIN model-checker is using PROMELA, a high-level language with data, but data values are instantiated (on bound domains) by the state exploration engines.

Many works have been done based on process algebra foundations, and have led to systems with a more developer-oriented specification language. The FDR2 tool [9] offers a high-level language for expressing CSP models, and an internal machine-readable dialect of CSP [10] using a specific expression language, more adapted to generate the models needed by the verification engines. The LTSA tool [11] uses Finite State Processes as an intermediate language (with processes and data parameters) for modelling concurrent Java programs. Another example of research showing goals close to ours makes use of Symbolic Transition Systems (STS) [12, 13], which are structures akin to our pNets. In the STSLib toolset, there is a dedicated specification language (with algebraic data types) for distributed components that are modelled by STS, themselves mapped to LOTOS programs that can be model-checked with CADP.

In all these cases, two important questions are: (1) how do you relate the programming language (or specification language) semantics with the internal model, and what properties are preserved by this mapping? (2) how do you transform your (parameterised) internal models into finite structures suitable for analysis (internal data structures of the verification engines, typically LTS, BDD, or various classes of automata…)?

Our proposal is different from previous approaches in the sense that we want a low-level model able to express various mechanisms for distributed systems, and that we do not limit ourselves to finite systems: we shall be able to define mappings to various classes of systems, finite or not. At the same time, the structure of our parameterised model is closer to the programming or specification language structure. Consequently, parameterised models are more compact, and easier to produce, than classical internal models.

Typically, our pNets model is lower-level than Lotos and Promela and more flexible for expressing different synchronisation mechanisms. On the other hand, it has no recursive constructs, in order to better control the finiteness of encodings.

Contribution This paper tries to answer these questions in the framework of distributed component systems. Toward this challenging perspective, we

develop a formal and parameterised behavioural model called *pNets*. We use this formalism to express models for ProActive distributed applications, Fractal components, and GCM distributed components. All our distributed models feature asynchronous calls with futures, which lowers latency while preserving a natural, data-flow oriented synchronisation.

One of the strong original aspects of this work is the focus put on non-functional properties, and the results we provide on the interleaving between functional and non-functional concerns. Thus, the programmer should be able to prove the correct behaviour of his distributed component system in the presence of evolution (or reconfiguration) of the system.

Structure of the paper In the next section, we recall the features of Fractal that are the most relevant to this study, describe the extensions proposed by the GCM model, and sketch the informal semantics of the GCM/ProActive implementation. In Section 3, we define formally our basic model, named pNets (this formalisation unifies and extends our previous publications in [14–17]) and recall the main properties of this model. In Section 4, we describe the model construction principles for four successive kinds of applications, namely active objects, hierarchical components, Fractal components with synchronous controllers and asynchronous GCM components with controllers. In Section 5, we describe the Vercors verification platform, and its application to a simple example, starting from the input specifications, through the model generation phase, to the verification of properties. We conclude with an analysis of perspectives of this work.

2 Context

2.1 Fractal, GCM and ProActive

The GCM [2] is a novel component model being defined by the European Network of Excellence CoreGrid and implemented by the EU project GridCOMP. The GCM is based on the Fractal Component Model [1] and extends it to address Grid concerns.

From Fractal, GCM inherits a hierarchical structure with strong separation of concerns between functional and non-functional behaviours, including, for example, life-cycle and binding management. GCM also inherits from Fractal introspection of components and reconfiguration capabilities.

Grids consider thousands of computers all over the world; for that, GCM extends Fractal using asynchronous method calls for dealing with latency. Grid applications usually have numerous similar components, so the GCM defines collective interfaces which ease design and implementation of such parallel components by providing synchronisation and distribution capacities. There are two kinds of collective interfaces in the GCM: multicast (client) and gathercast (server). Typically, a multicast interface is bound to the service interfaces of a number of parallel components, and a method call toward this interface is distributed, as well as its parameters, to several or all of them. GCM provides various policies for the request parameters that can be broadcast, or scattered, or distributed in a round-robin fashion; additional policies can be specified by the user. Symmetrically, gathercast interfaces are bound to a number of client components, and various synchronisation policies are provided. This treatment of collective communications provides a clear separation of concern between the programming of each component and the management of the application topology: within a component code, method calls are addressed simply to the component local interfaces. The management of bindings of clients (on a gathercast interface) or services (on a multicast interface) is separated from the functional code.

The GCM also allows the component controllers to be designed in the form of components, and benefit from such a design; moreover, the GCM specifies interfaces for the autonomic management and adaptation of components.

The Architecture Description Language (ADL) of both Fractal and the GCM is an XML-based format that contains both the structural definition of the system components (subcomponents, interfaces and bindings) and some deployment concerns. Deployment relies on *virtual nodes* that are an abstraction of the physical infrastructure on which the application will be deployed. The ADL only refers to an abstract architecture, and the mapping between the abstract architecture and a real one is given separately as a deployment descriptor.

2.2 A GCM reference implementation: GCM/ProActive

A GCM reference implementation is based on ProActive [18], an Open Source middleware implementing the ASP calculus [19, 20]. In this implementation, an active object is used to implement each primitive component and each composite membrane. Although composite components do not have functional code themselves, they have a membrane that encapsulates controllers and dispatches functional calls to inner

subcomponents. As a consequence, this implementation also inherits some constraints and properties with respect to the programming model:

- Components communicate through asynchronous method calls with transparent futures (placeholders for promised replies): a method call on a server interface adds a request in the server's *request queue*.
- Communication semantics uses a "rendezvous", ensuring the causal ordering of communications.
- Synchronisation between components is ensured with a data-flow synchronisation called *wait-by-necessity*: futures are first order objects that can be forwarded to any component in a non-blocking manner, execution is only blocked if the concrete value of the result is needed (accessed), and the result is still unavailable.
- There is no shared memory between components, and a single thread is available for each component.

Each primitive component is associated to an active object written by the programmer. Some methods of this active object are exported as the method of the component's interfaces. The active object managing a composite is generic and provided by the GCM/ProActive platform; it forwards the functional requests it receives to its subcomponents. Primitive component functionalities are addressed by the encapsulated active object. For primitive components, it is possible to define the order in which requests are served by writing a specific method called `runActivity()`; we call this the service policy. If no `runActivity()` is given, a default one implements a FIFO policy. Composite components always use a FIFO policy. Note that futures create some kinds of implicit return channels, which are only used to return one value to a component that might need it. One particularity of this approach is that it unifies the concept of component with the unit of distribution and parallelism: each primitive component represents the unit of distribution and is managed by a single thread. Composite components are also managed by their own thread and allocated separately, but there is no link between the location of a composite and the location of its subcomponents. One essential property of GCM/ProActive is that the global behaviour of a component system is totally independent of the physical localisation of components on a distributed architecture.

2.2.1 Life-cycle of GCM/ProActive components

GCM/ProActive implements the membrane of a composite as an active object; thus, it contains a unique request queue and a single service thread. The requests to its external server interfaces (including control requests) and from its internal client interfaces are dropped to its request queue. A graphical view of a composite is shown in Fig. 1.

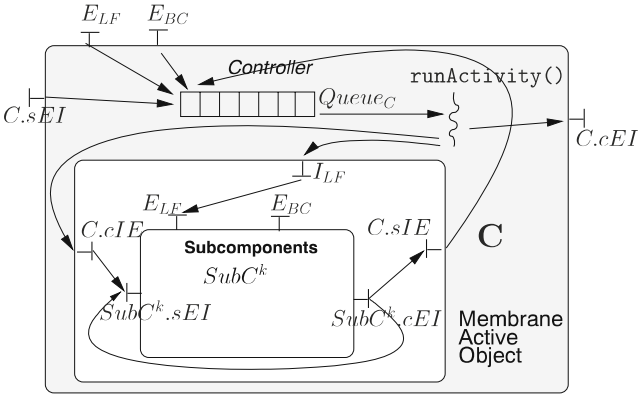


Fig. 1 *ProActive* composite component

Like in Fractal, when a component is stopped, only control requests are served. A component is started by invoking the non-functional request: `start()`. Because threads are non-interruptible in Java, a component necessarily finishes the request it is treating before being stopped. If a `runActivity()` method is specified by the programmer, the stop signal must be taken into account in this method. Note that a *stopped* component will not emit functional calls on its required interfaces, even if its subcomponents are active and send requests to its internal interfaces.

3 Theoretical model

In this section, we give the formal definition of our intermediate language that we call *parameterised Networks of Synchronised Automata (pNets)*. This language is not a new *calculus* in the tradition of theoretical computer science that gave birth to λ-calculus, π-calculus or σ-calculus, on which we would build new theories or new languages, nor is it a new process algebra endowed with syntax, semantics, and equivalences, that could be used to study new constructs for distributed computing. Rather, pNets give an intermediate and generic formalism intended to specify and synchronise the behaviour of a set of automata. We built this model with two goals: give a formal foundation to the model generation principles that we developed for various families of (distributed) component framework and build a model that would be more machine-oriented and serve as a versatile internal format for software tools, meaning it must be both expressive (from the

universality of synchronised LTSs) and compact (from the conciseness of symbolic graphs).

The synchronisation product introduced by Arnold and Nivat [5] is both simple and powerful because it directly addresses the core of the problem. One of the main advantages of using its high abstraction level is that almost all parallel operators (or interaction mechanisms) encountered so far in the process algebra literature become particular cases of a very general concept: synchronisation vectors. We structure the synchronisation vectors as parts of a *synchronisation network*. Contrary to synchronisation constraints, the network allows dynamic reconfigurations between different sets of synchronisation vectors through a *transducer* LTS. Our definition of the synchronisation product is semantically equivalent to the one given by Arnold and Nivat.

In the next step, we use Lin's [21] approach for adding parameters in the communications events of both transition systems and synchronisation networks. These communication events can be guarded with conditions on their parameters. Our agents can also be parameterised to encode sets of equivalent agents running in parallel. This leads us to the definition of pNets, that will later appear as a natural model of software systems. Indeed they correspond to the way developers specify or program these systems: the system structure is parameterised and described in a finite way (the code is finite), but a specific instance is determined at each execution, or even varies dynamically.

We now give the formal definitions of the model in two steps. In order for this article to be self-contained and with uniform notations, we first define LTSs, Nets and synchronisation product; these definitions are equivalent to those found in the literature. Then, we give the definitions of our parameterised structures (pLTS and pNet) and of their instantiations; their semantics are in terms of standard (infinite) LTS.

Notations In the following definitions, we extensively use indexed structures (maps or vectors) over some countable indexed sets. The indexes will usually be integers, bounded or not. When this is not ambiguous, we shall use abusive vocabulary and notations for sets, and typically write "indexed set over J" when formally we should speak of multisets, and still better write "mapping from J to the power set of $\mathcal{A}$".

We use uppercase letters $A, B, I, J\ldots$ to range over sets and lowercase letters $a, b, i, j\ldots$ to range over elements of the sets. We write $\tilde{A}_J$ for an indexed multiset of sets ($\tilde{A}_J = \langle A_j \rangle_{j\in J}$), and $\tilde{a}_J$ for an indexed multiset of elements ($\tilde{a}_J = \langle a_j \rangle_{j\in J}$), where J can possibly be infinite. For indexed sets of elements or sets, we say $\tilde{a}_J = \tilde{b}_I \Leftrightarrow J = I \wedge \forall j \in J, a_j = b_j$ (element-wise equality). We write $\langle a.\tilde{a}_J \rangle$ for the concatenation of an element a at the beginning of an indexed set, $\tilde{x}_J = \tilde{e}_J$ for an indexed set of equations ($\langle x_j = e_j \rangle_{j\in J}$), $e\{\tilde{x}_J \leftarrow \tilde{e}_J\}$ for the parallel substitution of variables $\tilde{x}_J$ by expressions $\tilde{e}_J$ within expression e.

As part of our abusive notation, we extensively, and sometimes implicitly, use the following definition for indexed set membership: $\tilde{a}_J \in \tilde{A}_J \Leftrightarrow \forall j \in J, a_j \in A_j$. Cartesian product is naturally extended to indexed sets so that the following is verified:

$$a_0 \in A_0 \wedge \tilde{a}_J \in \tilde{A}_J \Rightarrow \langle a_0.\tilde{a}_J \rangle \in \prod_{j\in\{0\}\cup J} A_j$$

We use the usual notions from (typed) term algebras: *operators*, *free variables*, *closed* and *open terms*, etc. Term algebras are endowed with a type system that includes at least a distinguished *Boolean* type and an *Action* type.

3.1 Networks of synchronised automata

We model the behaviour of a process as a LTS in a classical way [3]. The LTS transitions encode the actions that a process can perform in a given state.

Definition 1 LTS. A LTS is a tuple $(S, s_0, L, \rightarrow)$ where S (possibly infinite) is the set of states, $s_0 \in S$ is the initial state, L is the set of labels and $\rightarrow$ is the set of transitions $\rightarrow \subseteq SxLxS$. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

We define **Nets** in a form inspired by [5], that are used to synchronise a (potentially infinite) number of processes.

Definition 2 Net (Network of LTSs). Let Act be an action set. A Net is a tuple $\langle A_G, J, \tilde{O}_J, T \rangle$ where $A_G \subseteq Act$ is a set of global actions, J is a countable set of argument indexes, each index $j \in J$ is called a *hole* and is associated with a *sort* $O_j \subset Act$. The transducer T is a LTS $(S_T, s_{0T}, L_T, \rightarrow_T)$, and $L_T = \{\overrightarrow{v} = \langle a_g.\tilde{\alpha}_I \rangle.\ a_g \in A_G,\ I \subseteq J \wedge \forall i \in I, \alpha_i \in O_i\}$

Explanations Nets describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net. They are *transducers* in a sense similar to the Lotomaton expressions [22, 23]. A transducer in the Net is encoded as a LTS which labels are synchronisation vectors ($\overrightarrow{v}$), each describing one particular synchronisation between the actions (α_I) of different argument processes, generating a global action a_g. Each state of the transducer T corresponds to a given configuration of the network in which a given set of synchronisations is possible. Some of those

synchronisations can trigger a change of state in the transducer leading to a new configuration of the network; that is, it encodes a dynamic change on the configuration of the system.

We say that a Net is *static* when its transducer contains only one state. Note that each synchronisation vector can define a synchronisation between one, two or more actions from different arguments of the Net. When the synchronisation vector involves only one argument, its action can occur freely.

Definition 3 A *System* is a tree-like structure in which nodes are Nets and leaves are LTSs. At each node, a partial function maps holes to corresponding subsystems. A system is closed if all holes are mapped and open otherwise.

Definition 4 The *Sort* of a system is the set of actions that can be observed from outside the system. It is determined by its top-level node, L for a LTS, and A_G for a Net:

$$Sort(S, s_0, L, \rightarrow) = L \qquad Sort(\langle A_G, J, \tilde{O}_J, T\rangle) = A_G$$

As this is often the case in process algebras, sorts here are determined statically and are upper approximations of the set of actions that the system can effectively perform. The precision of this approximation depends naturally on the specific model generation procedure, but in most cases, an exact computation is not possible.

Building hierarchical Nets A Net is a generalised parallel operator. Complex systems are built by combining LTSs in a hierarchical manner using Nets at each level. There is a natural typing compatibility constraint for this construction, in terms of the sorts of the formal and actual parameters. The standard compatibility relation is Sort inclusion: a system *Sys* can be used as an actual argument of a Net at position j only if it agrees with the sort of the hole O_j ($Sort(Sys) \subseteq O_j$). Here, also, the compatibility relation may depend on the language or formalism that is modelled; for example, if actions represent Java-like method calls, the compatibility could take into account sub-typing.

Our behavioural objects being LTSs, and Nets being operators over LTSs, it is natural to give their semantics in terms of products over LTSs. The definition of the *synchronisation product* below defines the LTS representing any closed Net expression, computed in a bottom-up manner. It would be also possible to define a *symbolic* product over Nets that would reduce any *open* Net expression to a single Net, in the spirit of [22], but this is not necessary for our goals here.

Definition 5 Synchronisation product. Given an indexed set $\tilde{P}_J$ of LTSs $\tilde{P}_J = (\tilde{S}_J, \tilde{s_0}_J, \tilde{L}_J, \tilde{\rightarrow}_J)$, and a Net $\langle A_G, J, \tilde{O}_J, T = (S_T, s_{0_T}, L_T, \rightarrow_T)\rangle$, such that $\forall j \in J,\ L_j \subseteq O_j$, we construct the product LTS $(S,\ s_0,\ L,\ \rightarrow)$ where $S = \prod_{j\in\{T\}\cup J} S_j$, $s_0 = \langle s_{0_T}.\tilde{s}_{0_J}\rangle$, $L \subseteq A_G$, and the transition relation is defined as:

$$s \xrightarrow{l_t} s' \ \Leftrightarrow \ \begin{pmatrix} s = \langle s_t.\tilde{s}_J\rangle \wedge\ s' = \langle s'_t.\tilde{s}'_J\rangle \wedge \\ \exists s_t \xrightarrow{\langle l_t.\tilde{\alpha}_I\rangle} s'_t \in \rightarrow_T,\ \exists I \subseteq J,\ \forall i \in I, \\ \quad s_i \xrightarrow{\alpha_i} s'_i \in \ \rightarrow_i \ \wedge\ \forall j \in J\backslash I, s_j = s'_j \end{pmatrix}$$

3.2 Parameterised networks of synchronised automata

Next, we enrich the above definitions with parameters in the spirit of Symbolic Transition Graphs [21]. We start by giving the notion of parameterised actions. We leave unspecified here the constructors and operators of the action algebra; they will be defined together with the mapping of some specific formalism to pNets.

Definition 6 Parameterised actions. Let V be a set of names, $\mathcal{L}_{A,V}$ a term algebra built over V, including the constant action τ. We call $v \in V$ a parameter, and $a \in \mathcal{L}_{A,V}$ a parameterised action, $\mathcal{B}_{A,V}$ the set of boolean expressions (guards) over $\mathcal{L}_{A,V}$.

Example In Milner's *value-passing CCS* [3], the action algebra has constructors "`tau`", "`a`" for input actions, "`'a`" for output actions and "`a(x)`" for parameterised action. Then, "`'out(3)`" is a closed output action term, "`a(x,y)`" an open input action term with parameters `x` and `y` and "`x+y=3`" a guard.

Definition 7 pLTS. A parameterised LTS is a tuple $(V, S, s_0, L, \rightarrow)$ where:

- V is a finite set of parameters, from which we construct the term algebra $\mathcal{L}_{A,V}$.
- S is a set of states; to each state $s \in S$ is associated a finite indexed set of free variables $fv(s) = \tilde{x}_{J_s} \subseteq V$.
- $s_0 \in S$ is the initial state.
- L is the set of labels, $\rightarrow$ the transition relation $\rightarrow \subset S \times L \times S$.
- Labels have the form $l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}}\rangle$ such that if $s \xrightarrow{l} s'$, then:
  - α is a parameterised action, expressing a combination of inputs $iv(\alpha) \subseteq V$ (defining new

variables) and outputs $oe(\alpha)$ (using action expressions).
- $e_b \in \mathcal{B}_{A,V}$ is the *optional* guard.
- The variables $\tilde{x}_{J_{s'}}$ are assigned during the transition by the *optional* expressions $\tilde{e}_{J_{s'}}$

with the constraints: $fv(oe(\alpha)) \subseteq iv(\alpha) \cup \tilde{x}_{J_s}$ and $fv(e_b) \cup fv(\tilde{e}_{J_{s'}}) \subseteq iv(\alpha) \cup \tilde{x}_{J_s} \cup \tilde{x}_{J_{s'}}$.

Example Figure 2 is based on an implementation of the philosopher problem in ProActive. It represents the pLTS for the body behaviour of a Philo active object (see how we generate active object behaviour models in Section 4.1). The action alphabet used here reflects the active object communication schema: each remote request sent by the body has the form "$!dest.\texttt{request}(f, \mathcal{M}(\tilde{arg}))$", where *dest* is the remote reference, $\mathcal{M}$ is the method name, with parameters $\tilde{arg}$ and f is a future reference. More precisely, f is the identifier of the future proxy instance. Requests that do not require a response do not use a future proxy.

Definition 8 A *pNet* is a tuple $\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$, where: V is a set of parameters, $pA_G \subset \mathcal{L}_{A,V}$ is its set of (parameterised) external actions and J is a finite set of holes, each hole j being associated with (at most) a parameter $p_j \in V$ and with a sort $O_j \subset \mathcal{L}_{A,V}$. The transducer T is a LTS (S_T, s_{0T}, L_T, T_T), which transition labels ($\overrightarrow{v} \in L_T$) are synchronisation vectors of the form: $\overrightarrow{v} = \langle a_g, \{\alpha_t\}_{i\in I, t\in B_i}\rangle$ such that: $I \subseteq J \wedge B_i \subseteq \mathcal{D}om(p_i) \wedge \alpha_i \in O_i \wedge fv(\alpha_i) \subseteq V$.

Explanations Each hole in the pNet has a parameter p_j, expressing that this "parameterised hole" corresponds to as many actual arguments as necessary in a given instantiation of its parameter (we could have, without changing the expressivity, several parameters per hole). In other words, the parameterised holes express *parameterised topologies* of processes synchronised by a given Net. Each parameterised synchronisation vector in the transducer expresses a synchronisation between some instances ($\{t\}_{t\in B_i}$) of some of the pNet holes ($I \subseteq J$). The hole parameters being part of the variables of the action algebra, they can be used in communication and synchronisation between the processes.

A *static* pNet has a unique state, but it has state variables that encode some notion of internal memory that can influence the synchronisation. Static pNets have the nice property that they can be easily represented graphically. We have such graphics in previous publications to represent parameterised processes in the Autograph editor [24].

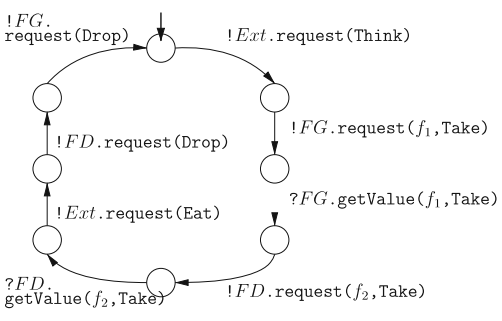
The sorts of our parameterised structures are sets of parameterised actions. This definition extends the simple sorts from Definition 4:

Definition 9 Parameterised sorts:

- The sort of a pLTS: $Sort(V, S, s_0, L, \rightarrow) = \{\alpha \mid \exists l \in L.\, l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}}\rangle\}$
- The sort of a pNet: $Sort\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle = pA_G$

Example The drawing in Fig. 3 shows a (static) pNet representing the classical philosophers problem, with two parameterised holes (indexed by the same variable k) for philosophers and forks. On the right-hand side are the corresponding elements of the formal pNet, in which we explicitly list the sort of the holes (O_{philo} and O_{Fork}), and where appear synchronisation vectors parameterised over the index k and the future ids f_1 and f_2.

Building hierarchical pNets Except from the occurrence of parameters in the structure of labels, the rest of the construction of complex systems as hierarchical pNet expressions is similar to the previous section, with the additional parameterisation of arguments: an actual (parameterised) argument of a pNet at position j is a



$PhiloRunActivityLTS = \langle V, S, s_0, L, \rightarrow\rangle$
with:

$V = \{f_1, f_2\}$

$S = \{s_i\}$, i ∈ [0:7]

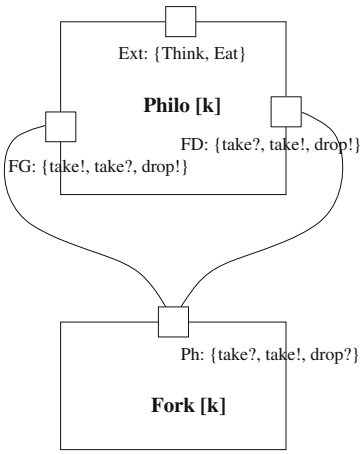
$L = \{$!Ext.request(Think), !Ext.request(Eat), !FG.request(f_1,Take), ?FG.getValue(f_1,Take), !FD.request(f_2,Take), ?FD.getValue(f_2,Take), !FG.request(Drop), !FD.request(Drop) $\}$

→ such that:

s_0 : !Ext.request(Think) → s_1,
s_1 : !FG.request(f_1,Take) → s_2
...

Fig. 2 Example of pLTS

Fig. 3 Example of pNet



$PhiloNet = \langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$ with:

$V = \{k, f_1, f_2\}$

pA_G = {Think(k), Eat(k), !TakeG(k), !TakeD(k), ?TakeG(k), ?TakeD(k), DropG!k, DropD!k}

J = {Philo, Fork}

$p_{Philo} = k, p_{Fork} = k$

O_{Philo} = {!*Ext*.request(Think), !*Ext*.request(Eat), !*FG*.request(f_1,Take), !*FD*.request(f_2,Take), ?*FG*.getValue(f_1,Take), ?*FD*.getValue(f_2,Take), !*FG*.request(Drop), !*FD*.request(Drop)}

O_{Fork} = {?*Ph*.request(f_1,Take), ?*Ph*.request(f_2,Take), !*Ph*.getValue(f_1,Take), !*Ph*.getValue(f_2,Take), ?*Ph*.request(Drop)}

This pNet is static, T has a unique state, and transitions with the following labels:
L_T = {
⟨Think(k), !Philo[k].*Ext*.request(Think)⟩
⟨Eat(k), !Philo[k].*Ext*.request(Eat)⟩
⟨!TakeG(k), !Philo[k].*FG*.request(f_1,Take), ?Fork[k].*Ph*.request(f_1,Take)⟩
⟨!TakeD(k), !Philo[k].*FD*.request(f_2,Take), ?Fork[k+1].*Ph*.request(f_2,Take)⟩
⟨?TakeG(k), ?Philo[k].*FG*.getValue(f_1,Take), !Fork[k].*Ph*.getValue(f_1,Take)⟩
⟨?TakeD(k), ?Philo[k].*FD*.getValue(f_2,Take), !Fork[k+1].*Ph*.getValue(f_2,Take)⟩
⟨DropG(k), !Philo[k].*FG*.request(Drop), ?Fork[k].*Ph*.request(Drop)⟩
⟨DropD(k), !Philo[k].*FD*.request(Drop), ?Fork[k+1].*Ph*.request(Drop)⟩ }

pair $\langle Sys, \mathcal{D}\rangle$, where *Sys* is a pNet (or pLTS) that agrees with the sort of the hole ($Sort(Sys) \subset O_j$), and $\mathcal{D}$ is the actual domain for the hole parameter p_j, i.e. denotes the set of similar arguments inserted in this hole.

We do not define a synchronisation product for pLTS that would give some kind of "early" or "symbolic" semantics of our generalised pNets. Instead, we define instantiations of the parameterised LTS and Nets, based on a (possibly infinite) domain for each variable.

Given a hierarchical pNet expression, and instantiation domains for all parameters in this expression, the definitions below allow us to construct a (non-parameterised) Net expression, by applying instantiation separately on each pLTS and each pNet in the expression. This can be performed both for closed or open pNet expressions, the result being, respectively, closed or open Net expressions. In the former, closed Net expressions can then be reduced to a single LTS (expressing the global behaviour) using the synchronous products in a bottom-up way.

Definition 10 pLTS Instantiation. Given a pLTS $P_p = \langle V, S_p, s_{0_p}, L_p, \rightarrow_p\rangle$, with $V = \tilde{x}_V$ and given a countable *domain* for each variable $\mathcal{D}_V = \{\mathcal{D}(x)\}_{x\in V}$, and an initial assignment ρ_0 for the variables of the initial state s_{0_p}, the instantiation $\Phi(P_p, \mathcal{D}_V)$ is a LTS $P = \langle S, s_0, L, \rightarrow\rangle$ such that:

- $S = \bigcup_{s_p\in S_p} \{s_p\{\tilde{x}_V \leftarrow \tilde{e}_V\} \mid \forall x \in V, \forall e_V \in \mathcal{D}(x)\}$,
- $s_0 = s_{0_p}\{fv(s_{0_p}) \leftarrow \rho_0(fv(s_0))\}$,
- L is the set of ground actions (i.e. closed terms) of the action algebra $\mathcal{L}_{A,V}$,
- $\rightarrow\ (\subseteq SxLxS) = \bigcup_{t\in\rightarrow_p} \Phi(t)$ is the union of instantiations of the parameterised transitions, built in the following way:
  let $t = s \xrightarrow{l_p=\langle\alpha, e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}}\rangle} s'_p$ be a transition, let $V_t = fv(s) \cup fv(\alpha) \cup fv(s')$ the free variables of t, and $\mathcal{D}_{V_t}$ their instantiation domains, then

$$\Phi(t) = \bigcup_{\tilde{e}_{V_t}\in\mathcal{D}_{V_t}} \left\{ \begin{array}{l} \text{if } \left(e_b\{\tilde{x}_{V_t} \leftarrow \tilde{e}_{V_t}\} = false\right) \text{ then } \emptyset \\ \text{otherwise} \\ \quad \text{let } \psi = \{\tilde{x}_{V_t} \leftarrow \tilde{e}_{V_t}\} \\ \quad \text{and } s'' = \text{if } \left(\exists j \in J_{s'}, x = x_j\right) \\ \qquad\qquad \text{then } s'\left\{x \leftarrow \psi(e_j)^\star\right\} \\ \qquad\qquad \text{else } s'\{x \leftarrow e_x\} \\ \quad \text{in } \left\{\psi(s) \xrightarrow{\psi(\alpha)} s''\right\} \end{array} \right\}$$

Apart from the proliferation of indexes, this definition is quite natural and straightforward; only the case when variables of the target state are assigned during the transition needs care (see * in the equation) because

the assigned open expressions $\tilde{e}_{J_{s'}}$ need themselves to be instantiated.

This operation has an upper-bound complexity that is exponential in the cardinality of the instantiation domains, in number of states and transitions.

Definition 11 pNet Instantiation. Given a pNet $N_p = \langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$, with the transducer $T = (S_T, s_{0T}, L_T, T_T)$, and given domains $\mathcal{D}_V$ for variables in V, the instantiation $\Phi(N_p, \mathcal{D}_V)$ is a Net $N = \langle A'_G, J', \tilde{O}'_{J'}, T'\rangle$, with $T' = \langle S_{T'}, s_{0T'}, L_{T'}, T_{T'}\rangle$ constructed in the following way:

1. Expand the parameterised holes: $J' = \Phi(J) = \biguplus_{j\in J}\mathcal{D}(p_j)$ where $\biguplus$ is a disjoint union (or concatenation) of sets; let $J'_j \subset J'$ be the part of J' corresponding to the expansion of hole number j.
2. Instantiate the sort of holes and the global sort:
   for $i \in J'_j$, build $\tilde{O}'_i = \bigcup_{a\in O_j}\Phi(a)$
   $A'_G = \bigcup_{a\in pA_G}\Phi(a)$
3. Instantiate the transducer:
   $S_{T'} = S_T$
   $s_{0T'} = s_{0T}$
   $L_{T'} = \bigcup_{\vec{v}\in L_T}\{\Phi(\vec{v})\}$ the expansion of the synchronisation vectors
   $T_{T'} = \bigcup_{(s,\vec{v},s')\in T_T}\{(s, a, s'), a \in \Phi(\vec{v})\}$ the expansion of the transition relation
   with $\Phi(\vec{v})$ computed by :

   let $\vec{v} = \langle a_g, \{\alpha_{i,t}\}_{i\in I, t\in B_i}\rangle$,
   let $V = fv(\vec{v})$,
   and $\mathcal{D}_V$ their instantiation domains,
   for each possible valuation $\tilde{e}_V$ of $\tilde{x} \in V$,
   (let $\phi = \{\tilde{x}_V \leftarrow \tilde{e}_V\}$ be the corresponding instantiation function,
   expand each parameterised action by
   $\Phi(\alpha_{j,t}) =$
   if $j \notin I$ then $\langle *, ..., *\rangle$
   else $\langle x_1, ..., x_{|J'_i|}\rangle$,
   with $x_k = *$ if $k \notin B_i$, else $\phi(\alpha_{j,t})$,
   build $\Phi(\phi, \vec{v})$ as a vector of cardinality $|J'|$
   as the concatenation of subvectors $x \in \Phi(\alpha_{j,t})$ for each hole $j \in J$),
   return $\Phi(\vec{v}) = \{\Phi(\phi, \vec{v})\}_{\{\tilde{e}_V\}}$

Naturally, even if the above definition does not suppose finiteness of the parameter domains, it will be used in practice with finite instantiation domains and finite vectors.

Example We give here a small instantiation of the philosopher system from Fig. 3:

$\Phi\left(PhiloNet, \mathcal{D}(k) = \{1, 2\}, \mathcal{D}(f_1) = \{1\}, \mathcal{D}(f_2) = \{2\}\right)$
=
$\langle A'_G, J', \tilde{O}'_{J'}, T'\rangle$ with:
$A'_G = \{$`Think(1), Think(2), Eat(1), !TakeG(1),...`$\}$
$J' = \{$`Philo, Philo, Fork, Fork`$\}$
$O\prime_{Philo^{(1)}} = \{$`!`$Ext$`.request(Think), !`$Ext$`.request(Eat), !`$FG$`.request(1,Take), ...`$\}$
$O\prime_{Philo^{(2)}} = \{$`!`$Ext$`.request(Think), !`$Ext$`.request(Eat), !`$FG$`.request(1,Take), ...`$\}$
...
$L_{T\prime} = \{$
`⟨ Think(1), !`Ext`.request(Think), *, *, * ⟩`
`⟨ Think(2), *, !`Ext`.request(Think), *, * ⟩`
`...`
`⟨ !takeG(1), !`FG`.request(1,Take), *,`
`?`Ph`.request(1,Take), * ⟩`
`⟨ !takeD(1), !`FD`.request(2,Take), *, *,`
`?`Ph`.request(2,Take) ⟩`
`... }`

Expressivity In [14], we gave examples of pNets representing various kinds of recursive functions: the "data flow" within an index family of pLTSs is expressed by an adequate indexing within the synchronisation vectors. However, one should note that this expressivity is gained from the properties of the indexes domains (here, integers with standard arithmetic): the pNets formal definition is (on purpose) separated from the data domain definition and does not provide by itself any formal expressivity result.

Another aspect of expressivity is the representation of classical patterns of distributed systems. We claim that pNets, used with simple (first-order) parameter domains, provide powerful and easy representations for our needs, including two-way or multi-way synchronisation, dynamic composition operators or dynamic creation/activation/orchestration of indexed families of processes, as will be exemplified in the following sections.

3.3 Data abstraction

The main interest of the instantiation mechanism defined so far is the ability to build specific domain instantiations with specific properties. In particular, if the instantiation domains are finite, and are built in such a way that they constitute abstract interpretations of the initial parameter domains, then the instantiated Net is finite. Moreover, if parameters were only used as value-passing variables in the original pNet (by contrast with parameters of the system topology), then we can apply

a result from Cleaveland and Riely [25] to justify the use of finite model-checking on our instantiated model:

Property 1 Let *Sys* be a closed pNet system, with parameters in V, (concrete) parameter domains $\mathcal{D}_V$ and abstract parameter domains $\mathcal{A}_V$, with the following hypothesis:

- Each $\mathcal{A}_v$ is an abstract interpretation[1] of the corresponding concrete domain $\mathcal{D}_v$.
- The domains of pNet holes parameters in *Sys* are unchanged by the abstraction.

Then, the abstraction preserves the *specification preorder*.

The *specification preorder* [25], or the better-known *testing preorder* [26], is closely related to safety and liveness properties. Given a system and a specification (set of properties), one can build a "most abstract" (finite) value interpretation relative to the specification, and try to establish its satisfaction. If this succeeds, the result is valid also for the concrete (potentially infinite) system; if it fails, one can select a more concrete (= more values) interpretation and repeat the analysis.

Unfortunately, the examples from this paper are too simple for giving a significant example of abstraction. Rather, let us use an example extracted from a previous case-study of our team modelling the Chilean electronic tax systems [27]. There we were manipulating invoice documents that could typically be described as structures $doct = \langle vendorid, invoiceid, date, content\rangle$ that would be checked by government services against $\langle vendorid, invoiceid\rangle$ records. In the case-study, we were using the abstract domain $doct = \langle vendorid \in [0..2], invoiceid \in [0..2]\rangle$ as an abstract interpretation preserving all safety properties involving, at most, two invoice documents.

In cases where the instantiated variables are parameters of the system topology, then the previous result does not apply. However, the same procedure can be used to build a finite model for one or more finite abstractions of the value domains. Even if this does not provide a proof of validity on the original system, it is still a valuable debugging tool. As an example, one could check safety properties involving Philo [1] and Fork [2] in the philosopher system, using an abstract domain for indexes defined as $\{\{1\}, \{2\}, \{others\}\}$. However, this will not prove that such a property holds for a system with an arbitrary number of philosophers.

[1] Cleaveland and Riely [25] was using a slightly relaxed condition called "galois insertions".

4 Behavioural models for distributed applications

In this section, we apply the pNets model to four examples, starting with distributed active objects. Then, we successively define a hierarchical component model and enrich it with non-functional controllers. We finally merge the previous concepts to get a modelisation of GCM/ProActive distributed components.

4.1 Active objects

The first application of pNets that we have published was for ProActive distributed applications, based on active objects, before the introduction of components. In [14, 15] we presented a methodology for generating behavioural models for ProActive, based on static analysis of the Java/ProActive code. This method is composed of two steps: first, the source code is analysed by classical compilation techniques, with special attention paid to tracking references to remote objects in the code and identifying remote method calls. This analysis produces a graph including the method call graph and some data-flow information. The second step consists in applying a set of structured operational semantics rules to the graph, computing the states and transitions of the behavioural model. The pNets model fits well in this context and allows us to build compact models, with a natural relation to the code structure: we associate a hierarchical pNet to each active object of the application and build a synchronisation network to represent the communication between them.

Figure 4 illustrates the structure of the pNets expressing an asynchronous communication between two active objects. A method call to a remote activity goes through a proxy that locally creates a "future" object, while the request goes to the remote request queue. The request arguments include the references to the caller and callee objects, but also to the future. Later, the request may eventually be served, and its result value will be sent back and used to update the future value.

The construction of the extended graphs by static analysis is technically difficult and fundamentally imprecise. Imprecision comes from classical reasons (having only static information about variables, types, etc.), but also from specific sources: it may not be decidable statically whether a variable references a local or a remote object. Furthermore, the middleware libraries include a lot of dynamic code generation, and the analysis would not be possible for code relying on reflexivity, classically used to manage some types of "dynamic topologies" in ProActive.

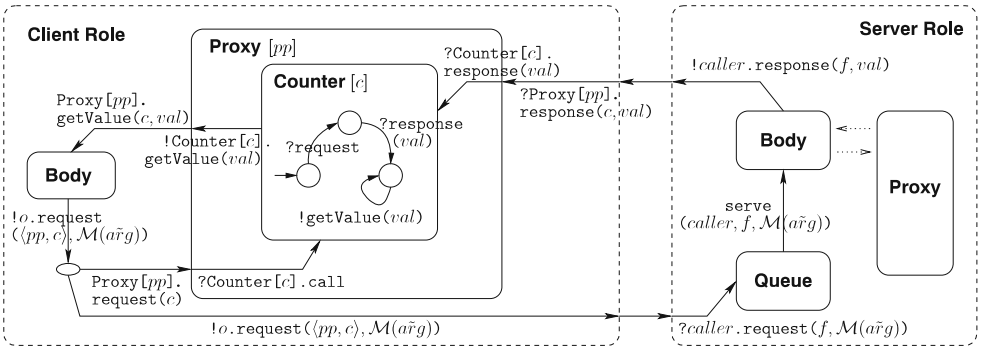


Fig. 4 Communication between two active objects

Nevertheless, for a reasonable subset of ProActive programs, we have the following result [15]:

Theorem 1 Finite pNet construction: *The analysis terminates, and (up to abstraction during analysis) each active object is modelled by a finite pNet hierarchy.*

More precisely, this result applies to most standard ProActive programs, with either FIFO or user-defined request selection policies, but with no usage of reflexivity in the Java code. It does not handle first-class futures, nor group communication, but extensions are currently studied. The strongest limitations come from the imprecision of the static analysis mentioned above, and from some difficulties when dealing with some of the Java constructs, like arrays of active objects.

4.2 Hierarchical components

Going from active objects to distributed and hierarchical components allows us to gain precision in the generated models. The most significant difference is that required interfaces are explicitly declared, and active objects are statically identified by components, so we always know whether a method call is local or remote. Moreover, the pNets's formalism expresses naturally the hierarchical structure of components.

To formalise the model generation for components, we give a definition of the structural information that is usually given through Architecture and Interface definition languages (ADL and IDL, respectively). This definition extends slightly those used in Fractal or in the GCM.

Definition 12 Component structure:

- A component C is a tuple $\langle V, \Sigma_V, \tilde{EI}, \xi\xi \rangle$, where V is a set of parameters, Σ_V a term algebra, $\tilde{EI}$ is the set of external interfaces of C, and ξ the content.
- An interface type $Ity = \langle \tilde{\mathcal{M}} \rangle$ is a set of methods $\mathcal{M} = \langle T, name, \tilde{A} \rangle$ with T its return type, and each $A = \langle T_A, name \rangle$ a typed argument.
- An interface is a tuple $Itf = \langle name, Ity, \kappa, \nu, \rho \rangle$, where Ity is its interface type, κ is the Fractal contingency (mandatory or optional), ν is the interface multiplicity, and ρ the interface role (either required or provided).
- The content of a composite component is a tuple $\xi = \langle \tilde{Itf}, \tilde{SubC}, \tilde{B} \rangle$, where $\tilde{Itf}$ is the set of internal interfaces, $\tilde{B}$ the set of bindings. $\tilde{SubC}$ is the set of parameterised subcomponents $SubC = \langle \nu, C \rangle$, with $\nu \in V$ a parameter and C a component.
- A binding B is a pair $\langle C_1.cItf, C_2.sItf \rangle$ with $C_i = \ self \mid sub\, C\left[expr \in \Sigma_V\right]$ identifies either the composite itself or one instance of a subcomponent, and $cItf$ is a client interface and $sItf$ is a server interface.

Note that we leave here undefined the content of a primitive component. It will depend on the framework and will be used to generate a pLTS representing the primitive behaviour. We also leave undefined the algebra Σ_V, which is used to build expressions for specifying indexes within the parameterised structure; it will depend on the domains used for the parameter V in a specific language.

From the information in a component structure, it is straightforward to generate a pNet representing the communication between the interfaces and the subcomponents, from the following elements:

- The pNet has one hole for each (parametric) subcomponent.
- The global actions pA_G and hole sorts $\tilde{O}_J$ of the pNets are sets of actions of the form $[!|?]\, C_i.Itf.\mathcal{M}(\tilde{arg})$ for invoking/serving a method $\mathcal{M}$ with each argument $arg \in \Sigma_{T_{arg},V}$.
- It has one parameterised synchronisation vector for each binding in $\tilde{B}$.

We have shown examples of proofs using such models in [28]. From now on, we have achieved a natural model generation for (parametric) hierarchical

systems, that can be compared with existing methods of other verification frameworks, e.g. CADP, μCRL or πADL. One important difference is that we have explicitly limited ourselves to (countable) static systems and use a property-preserving abstraction mechanism. Now, we build on this result to introduce some management and reconfiguration mechanisms in such a way that our verification methods still apply.

4.3 Hierarchical components + management interfaces = fractal

In the Fractal model, and in Fractal implementations, the ADL describes a static view of the architecture, and non-functional (NF) interfaces are used to control dynamically the evolution of the system. In this section, we define models for the Life-Cycle Controller (LF) and the Binding Controller (BC), in terms of pLTS generated from the component structure of the previous section.

Stopping a component in Fractal means that its functional activity is detained, while NF calls are still allowed in order to allow reconfiguring the component. This is modelled with an interceptor of all incoming calls. Then, depending on the components life-cycle (started or stopped), functional calls are allowed or not. Similarly, we only allow rebinding interfaces when the component is stopped.

A LF pLTS (see Fig. 5) is attached to each component. Control actions (`start`/`stop`) are synchronised with the parent component and with all of its subcomponents (note that this will not be the case for the asynchronous version), and status actions (`started`/`stopped`) are synchronised with the component's functional behaviour and with the BC because the BC may only allow rebinding of interfaces when stopped.

A BC pLTS (see Fig. 5) is attached to each interface. Control actions (`bind`/`unbind`) are synchronised up to the higher level (Fractal defines a white-box definition for NF actions) and with the affected interface; status actions (`bound`/`unbound`) are used to allow method calls $\mathcal{M}(\tilde{arg})$, to forward the call to the appropriate bound interface and to signal errors. The latter is a distinguished action $\mathcal{E}(unbound, C, Itf)$, visible to the higher level of hierarchy and triggered whenever a method call is performed over an unbound interface.

Alternatively, this could have been encoded using one state in the pNet transducer for each configuration of the bindings. However, this would require many transducer states, corresponding to all combinations of states of all controllers. Our approach is equivalent and more modular.

Note that we put external interface automata of a component in the next level of the hierarchy. This enables us to calculate the *controller* automaton of a component before knowing its environment. Thus, all the properties not involving external interfaces can be verified in a fully compositional manner.

By lack of space, we do not give here the detailed definition of the pNet expressing the synchronisation of the LF/BC controllers of a component with its functional behaviour, but we sketch its structure in Fig. 6. For synchronous Fractal components, the role of the interceptor is to synchronise incoming requests with the life-cycle state (either started or stopped actions) in order to restrict the allowed requests; allowed requests are synchronised with the inner part of the component (see Fig. 7).

In this drawing, the behaviour of subcomponents is represented by the box named $SubC^{\mathbf{k}}$. For each interface defined in the component's ADL description, a box encoding the behaviour of its internal (cII and sII) and external (cEI and sEI) views is incorporated. The dotted edges inside the boxes indicate a causality relation induced by the data flow through the box. Primitive components have a similar automaton without subcomponents and internal interfaces.

Building and using variants of this model The model construction is applied bottom-up through the hierarchy. The generated model is powerful enough to prove

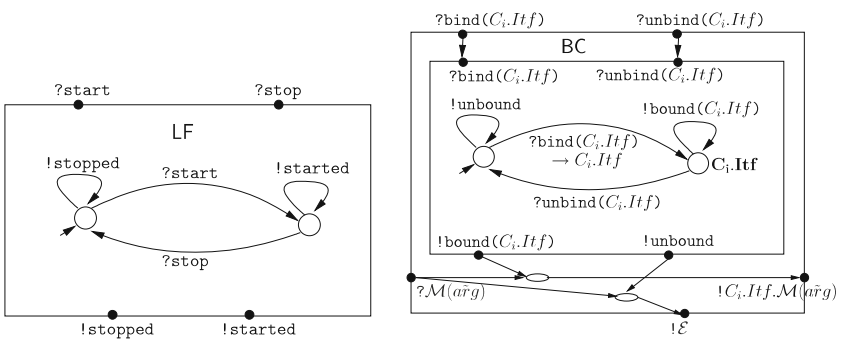


Fig. 5 pLTS of fractal life cycle and binding controllers

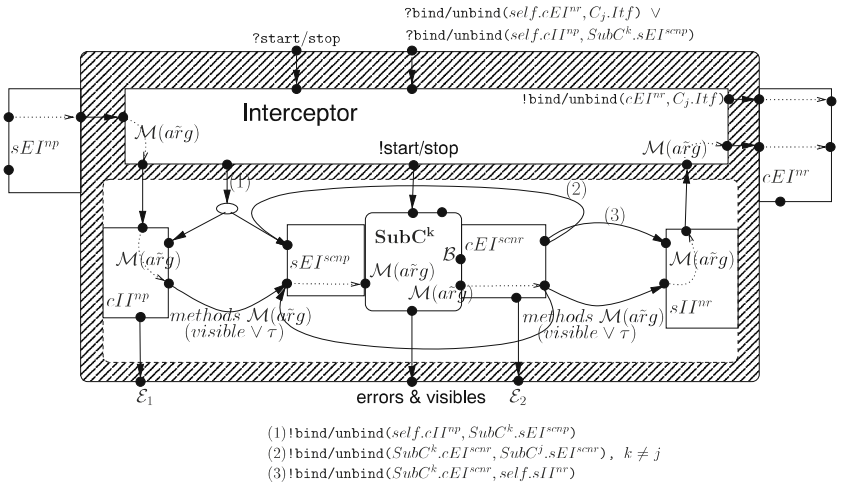


Fig. 6 Synchronisation pNet for a Fractal composite component

properties about deployment, normal behaviour or reconfiguration of a whole system. For pragmatic reasons, it is interesting to distinguish variants of this model in which only selected management actions are visible or authorised. We define the following variants:

- [*Static automaton*] This is the model in which all controllers are initialised in a "started" state, and all control actions are hidden. If the ADL was correct, then it should be equivalent (up to weak bisimulation) to the hierarchical component model (without controllers) from the previous section; otherwise, there will typically be reachable "unbound interface errors". It is used to check the normal behaviour of the system.
- [*Deployment automaton*] We define a *deployment sequence* for each composite as a sequence of control operations, expressed by an automaton, ending with a distinguished successful action $\surd$. We build a *non-deployed model* similar to the static model, but with controllers initialised in their `unbound` resp. `stopped` states. Then, the *deployment automaton* is the product of the non-deployed model with the deployment sequences. It allows us to check for correctness of deployment specifications, which is characterised by reachability of $\surd$.

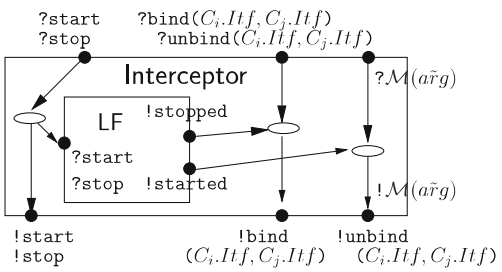


Fig. 7 Interceptor for synchronous Fractal components

- [*Reconfiguration models*] If we build the full model, then we can check properties relative to reconfiguration. This can be very costly because of the size of the action alphabet, so it can be refined by only keeping visible selected sets of control actions.

4.4 Distributed components: GCM/ProActive

In the Section 4.1 above, we have shown how to build the behaviour of ProActive activities; this corresponds exactly to the functional part of the behaviour of primitive components in our distributed implementation of Fractal. We now extend the model of Section 4.3 with this communication protocol in order to model GCM/ProActive components.

4.4.1 Primitive components

Let us recall the principle of asynchronous communication between two GCM/ProActive primitive components, inherited from ProActive (see Fig. 4). There, a method call on a client interface goes through a proxy that locally creates a "future" object, while the request goes to the request queue of the affected component. The request arguments include a reference to the future, together with a deep copy of the method's arguments; this is because there is no sharing between components. Later, the request may eventually be served, and its result value will be sent back to the future reference.

The body box in Fig. 4 represents the component's functional behaviour, and is itself modelled by a synchronisation network made from the synchronisation product of the `runActivity()` method's pLTS—ProActive's service policy—with the behaviour of service methods (methods defined by provided interfaces).

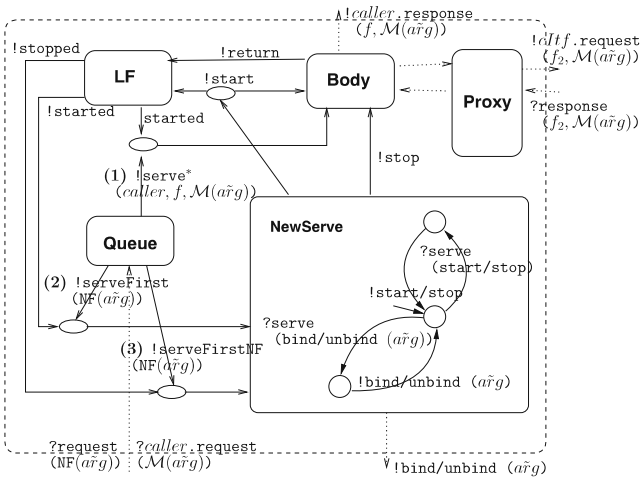


Fig. 8 Behaviour model for a GCM/ProActive primitive

In the model of a GCM/ProActive primitive component, we enrich the controller of the active object by adding two extra boxes, LF and NewServe, which correspond to the Interceptor in Fig. 6. The resulting pNet is drawn in Fig. 8. The body box is the only part that cannot be generated automatically from the ADL; it comes from the user-provided behaviour specification of the primitive (though its sort is fully specified).

NewServe implements the treatment of control requests. The action "start" fires the process representing the method `runActivity()` in the body. "Stop" triggers the `!stop` synchronisation with body (Fig. 8). This synchronisation should eventually lead to the termination of the `runActivity()` method (`!return` synchronisation). In the GCM/ProActive implementation, this is done through setting the state variable `isActive` to false, which should eventually cause the `runActivity()` method to finish, only then the component is considered to be stopped. Note that this may depend on the programmer's implementation of the `runActivity()` method, so it is worth verifying in the generated model!

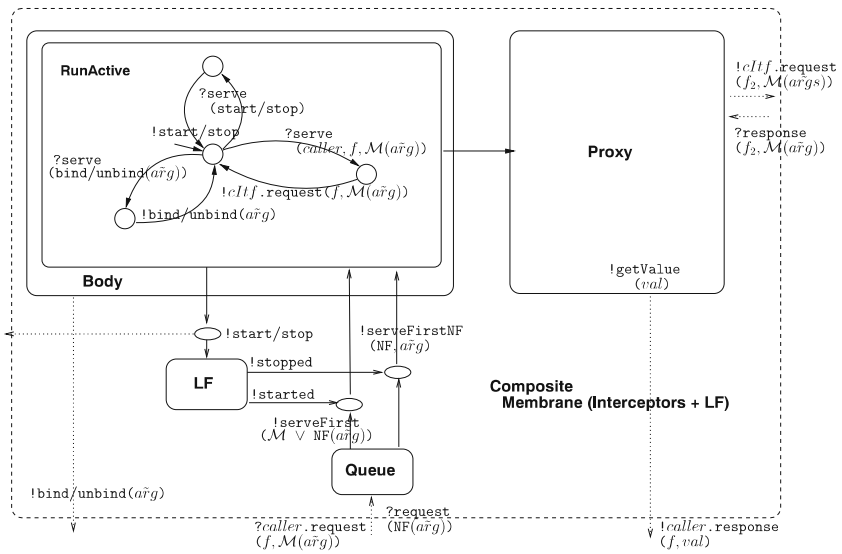
The queue box can perform three actions: (1) serve the first functional method corresponding to the `Serve` API primitive used in the body code, (2) serve a control method only at the head of the queue and (3) serve only control methods in FIFO order, bypassing the functional ones.

4.4.2 Composites components

A composite membrane in GCM/ProActive is an active object. When started, it serves functional or control methods in FIFO order, forwarding method calls between internal and external functional interfaces. When stopped, it serves only control requests.

Figure 9 shows the model of the membrane, that is similar to the *interceptor* from Fig. 6, though more complex. The membrane model is created from the description of the composite (given by the ADL). The proxy in Fig. 9 is the same as the one presented in Fig. 4. In this case, the proxy is in charge of forwarding the value of the future by receiving the value in action `?response(..)` and forwarding in action `!response(..)`. Since the method calls include the reference of the future in the arguments, future updates can be addressed directly to the caller immediately before in the chain. Consequently, like in the implementation, the future update would not be affected in case of a rebinding or a change in the life-cycle status

Fig. 9 Behaviour of a composite membrane



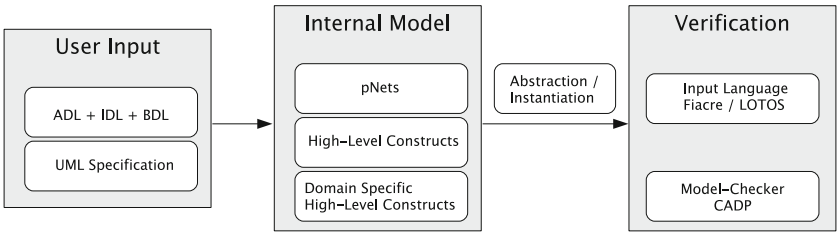


Fig. 10 The VERCORS architecture

of the components. Our model is expressive enough to reflect this property.

The modelisation here does not handle the mechanisms for *first-class futures*, which require specific controllers for storing and updating chains of future proxies through several components. We are working on this extension. This is important both for reflecting realistic applications that use this mechanism for efficiency and because it has significant behavioural impact: deadlocks may be different when you allow first class futures.

5 Platform overview

We present below a high-level view of the Vercors platform and the properties we are able to verify; the interested reader could refer to [17] for further details. Our platform comprises several tools for assisting the verification process. Rather than creating a new model-checker, we implement our model-generation methods in a way that they efficiently integrate with existing state-of-the-art tools for checking component specifications based on the models of Section 4. The platform is presented through the classical problem of a bound buffer with one consumer and one producer.

Figure 10 gives a snapshot of the platform. In the next subsections, we shall describe in detail its three parts: the input from the user (Section 5.1), the behavioural model (Section 5.2), and the verification of properties (Section 5.3). We illustrate our platform through the formal verification of the previously outlined case-study.

5.1 User input

For automatically building the behavioural model, we take a two-fold approach: (1) the architecture and hierarchy information are extracted from the ADL (and IDL) and (2) each of the primitive component's functional behaviour is specified by the user in an automata-based language which we call Behavioural Description Language (BDL).

Figure 11 shows an example of a producer consumer system. Both the producer and the consumer produce/consume one element at a time. Additionally, the buffer emits an alarm through its interface I_{alarm}, when the buffer is full.

The XML description of the ADL of the producer–consumer example is shown in Fig. 12. It specifies that the system is composed of the composite BufferSystem (line 6), itself described in a separate file (`components/BufferSystem.fractal`), and the primitive Alarm, the implementation of which is the Java class `components.Alarm` (line 15). The BufferSystem receives a paramater (three in our example, line 7) used to initialise the component with the maximal size of the buffer. The BufferSystem also requires an interface named `alarm` of type `components.AlarmInterface` (lines 8 and 9). Alarm provides an interface named `alarm` of type `components.AlarmInterface` (lines 13

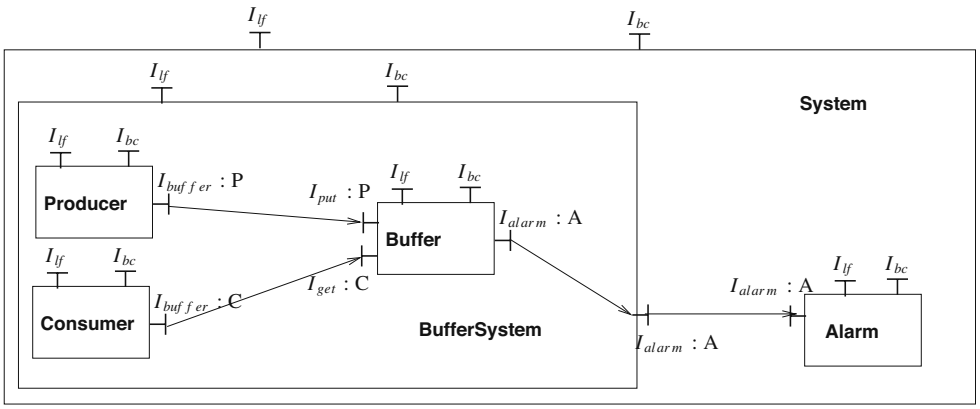


Fig. 11 Consumer–producer example

System.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE .... >

<definition name="components.System">

   <component name="BufferSystem"
         definition="components.BufferSystem(3)">
      <interface name="alarm" role="client"
            signature="components.AlarmInterface"/>
   </component>

   <component name="Alarm">
      <interface name="alarm" role="server"
            signature="components.AlarmInterface"/>
      <content class="components.Alarm">
         <behaviour file=''AlarmBehav''
               format=''FC2Param''/>
      </content>
   </component>

   <binding client="BufferSystem.alarm"
        server="Alarm.alarm"/>
</definition>
```

Fig. 12 System ADL

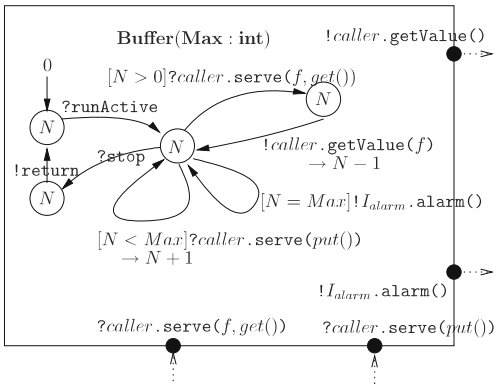


Fig. 13 Buffer behaviour (provided by user)

and 14). Then, interface signatures are given with the Fractal Interface Definition Language (IDL). In the implementations we consider, this definition is given by Java interfaces describing the signatures of the methods of each component interface. Analysing the ADL and the IDL, we are able to build the behavioural model with asynchronous and non-functional controllers of Section 4.4.

Finally, the functional behaviour is given by a BDL, in this case in pNets. An example of a behavioural specification of the `Buffer` is given in Fig. 13. The abstract specification does not consider the values of the elements, but only the amount of elements stored. Therefore, the parameterised automaton has a variable N representing the number of elements stored in the buffer, and the transitions have guards with expressions related to this variable and to a constant Max. In the example, the buffer is instantiated with $Max = 3$ as set in line 7, Fig. 12. Other parameters are: *caller*, representing the reference to the activity (component) that invoked the method call, and f, representing the identifier of the future that is used to send back the response. The buffer also invokes methods on its client interface I_{alarm} in case the buffer is full (action $!I_{alarm}.\texttt{alarm}()$).

Although pNets can be used as a BDL, it is convenient to give a higher-level language to non-expert users. In this vein, we also developed a tool called CTTool [29], using UML2 statemachines diagrams to express pLTSs, and a variant of UML2 component structures to specify the system architecture (but only in the static case). We also plan to provide a textual specification language that would smoothly integrate architecture and behaviour specifications for GCM applications, but this is still in progress.

5.2 Internal model

We first automatically build the behavioural model in pNets seen in Section 4. This is done by ADL2N, which is a tool written in Java for generating the behavioural models of Fractal components by analysing the system's ADL and IDL (see Section 4.2).

We also specified a model for Fractal's binding and life-cycle controllers. Those two controllers allow us to model the deployment and some basic reconfigurations of the system. In our case, checking the safeness of these can be done statically by building the *Static*, *Deployment* or *Reconfiguration* automata of Section 4.3.

In practice, the user of ADL2N uses a GUI to specify at the same time the methods that will be visible, the arguments that are significant and their finite instantiation. The visibility of methods and the abstraction (see Section 3.3) depend on the formulas to be checked. Although it should be possible to infer safe abstractions given a set of formulas, for the moment, it is up to the user to provide finite abstractions of the data domains. The output of ADL2N is the pNets behavioural model of Section 4.3 with the above abstractions and with the selected actions hidden.

5.3 Verification

In the current toolset, we only interface with finite-state model-checkers and, namely, with the evaluator model-checker from the CADP toolset, that features a very efficient check of branching-time logics, together with on-the-fly generation, cluster-based distributed state-

generation, tau-confluence reduction, etc. We give here examples of verification for various usage scenarios.

Deployment In GCM/ProActive, method calls are asynchronous, and there may be delays between the request for a non-functional method and its treatment. So checking the execution of a control operation must be based on the observation of its application on the component, rather than the arrival of the request.

One of the interesting properties is that the start operation, which is hierarchical, occurs during the deployment; i.e. that the component and all its subcomponents are at some point started. This property can be expressed as the (inevitable) reachability of the start signal in the static automaton of `System`, for all the possible executions, where `name = {System, BufferSystem, Alarm, Buffer, Consumer, Producer}`. This can be translated into a μ-calculus formula and verified in CADP.

Pure-functional properties The classical interesting properties concern the behaviour of the system after its deployment, at least while there are no reconfigurations. For instance, in the example, we would like to prove that a request for an element from the queue is eventually served, i.e. that the element is eventually obtained. This is proved to be true in CADP by model-checking the global state-space.

Functional properties under reconfigurations Our approach enables the verification of properties not only after a correct deployment, but also after and during reconfigurations. For instance, the pure-functional property above becomes false if we stop the producer since, at some point, the buffer will be empty, and the consumer will be blocked waiting for an element. However, if the producer is restarted, the consumer will eventually receive an element and the property should become true again. In other words, we can check that, if the consumer requests an element, and then the producer is stopped, if the producer is started again, the consumer will get the element requested.

For proving this kind of property, the static automaton is not sufficient; we need a behavioural model containing the required reconfiguration operations. We add to the component network a *reconfiguration controller* (Fig. 14): its initial state corresponds to the deployment phase and the next state corresponds to the rest of the life-cycle in which reconfigurations are enabled. This state change is fired by the successful termination of the deployment ($\surd$). For the property stated above, the reconfigurations `?stop(Producer)` and `?start(Producer)` are left visible.

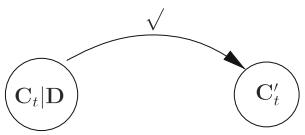


Fig. 14 Synchronisation product supporting further reconfigurations

Asynchronous behaviour properties Let us now focus on a property specific to the asynchronous aspect of the component model. The communication mechanism in GCM/ProActive allows any future, once obtained, to be updated with the associated value, provided that the corresponding method is served and terminates correctly; binds, unbinds or stops operation cannot prevent this. For example, if the consumer is unbound after a request, it gets anyway the response, even if the link is then unbound or the component stopped. We are able to verify this in our behavioural models.

6 Conclusion and perspectives

This article defines the pNets formalism, a parameterised and hierarchical extension of LTSs. pNets have a tree-structure in terms of networks of synchronisation vectors, and a very high expressivity through the use of parameters at both LTS and network levels. This formalism is used to represent the behavioural semantics of distributed systems. It provides a compact and well-defined intermediate format for connecting code analysers or code generators with model-checking or equivalence engines.

In addition to the formal definition of pNets, our contribution is:

- Four scenarios demonstrating the usage of pNets. We generate behavioural models for active objects, hierarchical components, hierarchical components with non-functional controllers and finally asynchronous hierarchical components with non-functional controllers.
- A short description of our verification platform Vercors, in which we use pNets as the pivot format for analysis, abstraction, verification and code-generation tools. We show the results of model construction and analysis of temporal logic properties for a simple case-study.

The pNets format is lower-level, and more versatile, than other models used in existing verification toolsets. Many tools rely on specific synchronisation and communication mechanisms, like the LOTOS-like parallelism in the CADP toolset, channels in Promela or Petri nets in other cases. In contrast, the low-level

primitives of pNets (LTS + synchronisation vectors) are able to represent many possible mechanisms, as demonstrated by the four applications in this article.

Another important trade-off is between parameterised representations (close to developers code) and lower-level explicit-state encodings that are required by model-checkers. We argue that the pNets model allows for finite and compact representation of systems, expressive enough to capture a large family of behavioural properties of both synchronous and asynchronous applications.

The Vercors platform (editor, generation, instantiation and conversion tools) and a large-scale case-study are available at our website.[2] These tools currently allow to build behavioural models for synchronous Fractal components with partial support for non-functional controllers. They are interfaced with the explicit-state verification toolset CADP.

We are currently working on the controller generation for the GCM/ProActive asynchronous components, including the handling of multicast/gathercast communications, of transparent futures and of component reconfiguration. A specific concern is the encoding of request queues; a direct representation with pNets is possible but would be very expensive in term of state/transition complexity. We are looking for a specific parametric representation coupled with a specialised "infinite-state" engine.

Our main application context is the GCM component model and its reference implementation within the Java/ProActive library. However, static analysis of Java/ProActive code is intrinsically imprecise, making the generation of pNet models difficult. We are working on a specification language integrating architectural and behavioural views, with high-level constructs for system reconfiguration, and for Grid specific features like collective interface policies and parameterised component topologies. This language will be used as an input for the Vercors platform, but also for tools that will generate "correct by construction" Java code.

References

1. Bruneton E, Coupaye T, Leclercp M, Quema V, Stefani J (2004) An open component model and its support in java. In: 7th int symp on component-based software engineering (CBSE-7), LNCS, vol 3054. Springer
2. CoreGRID, Programming Model Institute (2006) Basic features of the grid component model (assessed). Technical report, Deliverable D.PM.04. http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf
3. Milner R (1989) Communication and concurrency. Prentice Hall, Englewood Cliffs ISBN 0-13-114984-9
4. Bergstra J, Pose A, Smolka S (2001) Handbook of process algebra. North-Holland, Amsterdam
5. Arnold A (1994) Finite transition systems. Semantics of communicating sytems. Prentice-Hall, Englewood Cliffs
6. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes. Inf Comput 100(1):1–77
7. Garavel H, Lang F, Mateescu R, Serve W (2007) CADP 2006: a toolbox for the construction and analysis of distrbuted processes. In: CAV 2007 conference. Berlin, Germany
8. Garavel H, Lang F (2002) NTIF: a general symbolic model for communicating sequential processes with data. In: Proceedings of FORTE'02 (Houston), LNCS, vol 2529. Springer
9. Roscoe A (1994) Model-checking CSP. In: A classical mind, essays in honour of C.A.R. Hoare. Prentice-Hall, Englewood Cliffs
10. Scattergood J (1998) The semantics and implementation of machine-readable CSP. PhD thesis, Oxford Un. Computing Laboratory
11. Magee J, Kramer J (2006) Concurrency: state models and java programs, 2nd edn. Wiley, New York
12. Poizat P, Royer J, Salaun G (2006) Bounded analysis and decomposition for behavioural descriptions of components. In: FMOODS, LNCS, vol 4037. Springer
13. Poizat P, Royer J (2006) A formal architectural description language based on transition systems and modal logic. J Univers Comput Sci 12(12):1741–1782
14. Barros T, Boulifa R, Madelaine E (2004) Parameterized models for distributed Java objects. In: Forte'04 conference. LNCS, vol 3235. Springer, Madrid
15. Boulifa R (2004) Génération de modèles comportementaux des applications réparties. PhD thesis, University of Nice - Sophia Antipolis – UFR Sciences
16. Barros T, Henrio L, Madelaine E (2005) Behavioural models for hierarchical components. In: Godefroid P (ed) Model checking software, 12th int SPIN workshop, LNCS, vol 3639. Springer, San Francisco
17. Barros T (2005) Formal specification and verification of distributed component systems. PhD thesis, University of Nice - Sophia Antipolis
18. Caromel D, Delbé C, di Costanzo A, Leyton M (2006) ProActive: an integrated platform for programming and running applications on grids and P2P systems. Comput Methods Sci Technol 12(1):69–77
19. Caromel D, Henrio L, Serpette B (2004) Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, New York, pp 123–134
20. Caromel D, Henrio L (2005) A theory of distributed object. Springer, Heidelberg
21. Lin H (1996) Symbolic transition graph with assignment. In: Montanari U, Sassone V (eds) CONCUR '96, LNCS, vol 1119. Pisa, Italy
22. Lakas A (1996) Les Transformations Lotomaton: une contribution à la pré-implémentation des systèmes Lotos. Ph.D. thesis, Univ. Paris VI
23. Najm E, Lakas A, Serouchni A, Madelaine E, de Simone R (1992) ALTO: an interactive transformation tool for LOTOS and LOTOMATON. In: Bolognesi T, Brinksma E, Vissers C (eds) Third lotosphere workshop and seminar, Pisa
24. Madelaine E (1992) Verification tools from the CONCUR project. In: Rozenberg G (ed) EATCS Bull, vol 47. B. Rovan, Bratislava

[2] http://www-sop.inria.fr/oasis/Vercors.

25. Cleaveland R, Riely J (1994) Testing-based abstractions for value-passing systems. In: CONCUR'94, LNCS, vol 836. Springer, Heidelberg
26. Cleaveland R, Hennessy M (1993) Testing equivalence as a bisimulation equivalence. Form Asp Comput 5:1–20
27. Attali I, Barros T, Madelaine E (2004) Formalisation and proofs of the chilean electronic invoices system. In: Proc. of the XXIV international conference of the Chilean computer science society (SCCC'04). IEEE, Arica
28. Barros T, Cansado A, Madelaine E, Rivera M (2006) Model checking distributed components: the Vercors platform. In: 3rd workshop on formal aspects of component systems. ENTCS, Prague
29. Ahumada S, Apvrille L, Barros T, Cansado A, Madelaine E, Salageanu E (2007) Specifying fractal and GCM components With UML. In: Proc of the XXVI international conference of the Chilean computer science society (SCCC'07). IEEE, Iquique

Chapitre 5

Tool platform

5.1 Summary

The paper included in this chapter [C-09] was presented in September 2008 at the FMCO symposium. Based on the formal model presented in the previous chapter, it recalls the construction of pNets models for distributed hierarchical components, and extends it for two important features of GCM components, namely *first-class futures* and *collective communication interfaces* (we have summarized these contributions in the previous chapter, section 4.1). It also gives a synthetic view on our tool platform Vercors, with a focus on the Component Architecture Editor, and the tools for generation and manipulation of pNets. This work was developed between 2005 and 2008 in collaboration with Tomás Barros, Antonio Cansado and Ludovic Henrio, but also with the internships of Alejandro Vera, Marcela Rivera, Emil Salageanu [83, 82], Pablo Valenzuela [33], and Krzysztof Nirski.

As mentioned in the conclusion of the previous chapter, it quickly became mandatory to run significant experiments, starting with proof-of-concept prototypes, and progressively assembling them, together with external tools when available, in a coherent platform.

We started with Tomás Barros, developing tools to support the heart of the pNets model, using the FC2 format inherited from my previous work in the Meije team. FC2 provided us with a concrete syntax to express hierarchical process structures, and more importantly, parameterized synchronization vectors. Tomás developed the tools **FC2instantiate** and **FC2toExp** for instantiating parameterized FC2 structures into finite FC2, and for translating finite FC2 structures into Exp files, that is the format for synchronization vectors of the CADP toolset. At that point, we had no tool for automatically translating pNets models into FC2 syntax, and we had to write parameterized FC2 format by hand, that was somewhat difficult. But this was enough to run our first large-scale use-case, based on a specification of the *Chilean Electronic Invoices System* (see section 7.1), that we published at SCCC'04 [C-04b,R-04].

At this point we had a good support for state-space generation and for model-checking, using CADP [47], and a usable syntax for encoding pNets models in FC2 syntax. But writing these by hand was not reasonable, and we started to look for tools that would be convenient for system specification (both behaviours and parallel structure), and from which we could generate FC2 code. The requirements in this quest were manifolded. We wanted :

– a formalism expressive enough to cover all aspects of pNets, from simple data manipulation within transition systems, to hierarchical parallel pro-

cesses with various communication and synchronization artifacts,
– a formalism intuitive and accessible to non specialists,
– an easy expression of our distributed objects and components semantics.

Our first ideas, and early experiments, were based on UML2.0, in which the existing *Activity*, *State-Machine*, and *Component Structure* diagrams were not too far from our needs. We developed **CTTool** [81] from these ideas, based on the **TTool** environment of Ludovic Apvrille [7], and defined a custom version of the *Component Structure* meta model adapted to our needs. We published this work in [C-06,C-07a] and [82], and the tool was used extensively in the CoCome Case-study [J-08], as illustrated in Figure 5.1.

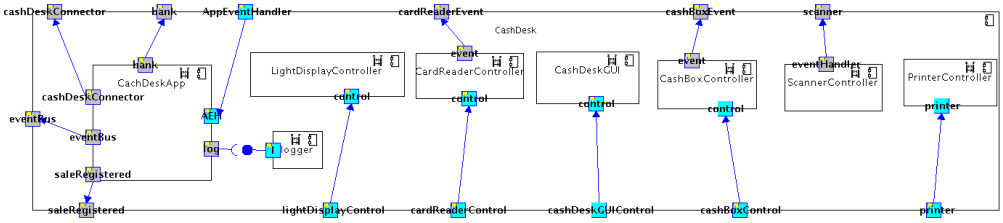


Fig. 5.1 – A CTTool drawing from the CoCoME case-study

CTTool was directly producing Lotos code suitable for verification in the CADP toolset, but this was not a good idea, because we had no direct control on the semantics of the graphical constructions, and indeed, we were not able to implement correctly all pNets constructs. Another serious drawback was that the structure of UML component structures are significantly different from GCM components (see [C-07a,W-06]) ; defining a GCM "profile" based on UML2.0 would have been possible, but the differences from original component structures would have been more significant than the similarities.

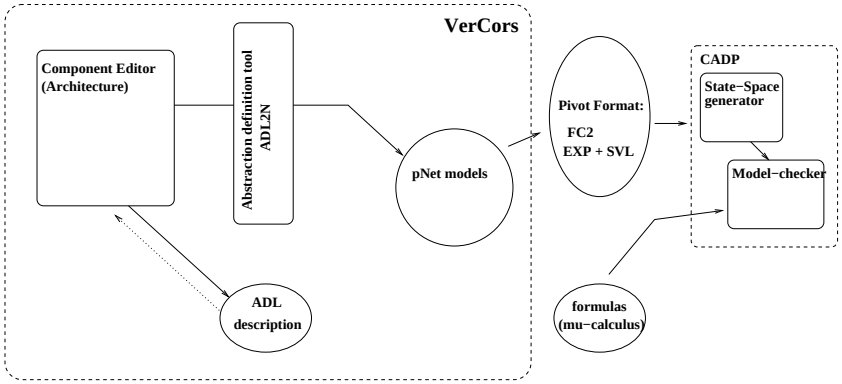


Fig. 5.2 – Initial architecture of VerCors tool set

At this point, it became clear that it would be much more convenient to have our own graphical language for specifying GCM component architectures. At the same time, several platforms were available for developing well integrated Eclipse-based environments, and we started to build what is now the **VerCors** platform, in particular with support from our ACI collaborative project FIACRE (see section 9.4). The distributed version of VerCors, as described in the FMCO paper (pages FMCO :18-22), now includes the graphical editor for GCM components **VCE**, able to read and write GCM ADL descriptions ; the **ADL2N** tool, that builds pNets structures from component drawings, including (abstract) data parameters ; and

connexion to the CADP toolset through the FC2 and EXP formats.

Figure 5.2 illustrates the architecture of the first version of the VerCors platform. On the left is the VerCors Component Editor VCE, which is able to read and write component architecture descriptions (Fractal ADL). On the right is the CADP toolset ; in the current state of VerCors, running the verification tools is left to the user, and the formulas have to be written directly in the model-checker language.

Since this publication at FMCO, we have concentrated our efforts on : 1) the replacement of FC2 by the `Fiacre` format, 2) experimentations with new model-checking engines, including algorithms for representing and analyzing processes communicating over unbounded channel ; and distributed model-generation and model-checking engines. We detail these points below.

The `Fiacre` format All verification platforms and model-checking engines have their own input language(s). CADP accepts a number of different languages, including Lotos, ELotos, CCS, CSP, mCRL, and none of these languages were directly suitable to encode our pNets systems, because each of them have their specific parallelism and synchronization constructs and semantics. We were looking in fact for a lower level formalism, that would easily encode correctly and concisely the pNets systems.

Within the FIACRE project, we contributed to the development of the `Fiacre` format, which is a pivot format in the development platforms created by the Topcased (www.topcased.org) and the OpenEmbeDD (openembedd.org) collaborative projects. `Fiacre` is a textual format for representing communicating automata, including typed channels, a set of simple data types, sequential control constructs, and parallel constructs for building hierarchical processes. `Fiacre` can be used as an input language of CADP, through the use of a Fiacre to Lotos compiler. `Fiacre` is not able to represent directly all constructs of the pNets model, in particular when multi-point synchronization is involved. However, it is a good choice of intermediate format between Vercors and CADP in the current situation, especially for encoding models of pure (parameterized) LTSs, or simple compositions of LTSs. For more complex synchronization structures, we can use the EXP format of CADP, for expressing our synchronization vectors.

It should be noted that such a choice is always a compromise between the complexity of representation of our models, the ease of translation, but also of the precision of the inverse translation, when one needs to translate diagnoses of the analysis engine into the original model.

An important tool currently missing in the Vercors platform is a translator between the pNets model (generated from textual or graphical editors), to a pivot format, or combination of formats, that would automate the interface with the model-checking toolsets. In some sense, what we seek is a syntactic representation of the pNets model, in term of Fiacre/Exp processes. We have started, with Adel Bouchakhchoukha [26] to define this translation, and we have experimented in our recent use-cases with the structuring of the pivot format, with in mind its utilization for compositional and distributed model-checking (see chapter 7). The translator still has to be implemented, and integrated in VerCors. This will be one of our short term priorities, with the following goal :

The translator from pNets to the pivot format in VerCors should :

- be incorporated as an Eclipse plugin, to be activated directly from the editors,
- be “invertible”, in the sense that diagnoses from the model-checkers have to be translated back into the original user-level syntax, and eventually dis-

played in the editors,
– hide as much as possible the complexity of the underlying model, so that non specialists users can run the model-checkers.

Figure 5.3 shows the architecture of the future VerCors version. The main differences with Figure 5.2 are : the addition of behaviour and formulas editors ; the replacement of FC2 by the Fiacre formalism ; the possibility to plug in new model-checkers ; the addition of "backward" flows (dashed arrows) expressing the translation of diagnostics back to the user-level formalisms and editors.

Note that it may be the case that new model-checkers have different input formalisms, and not accept Fiacre/Exp programs. Still we want to produce such inputs from pNets models, and be able to reuse the rest of the tool chain in a coherent way.

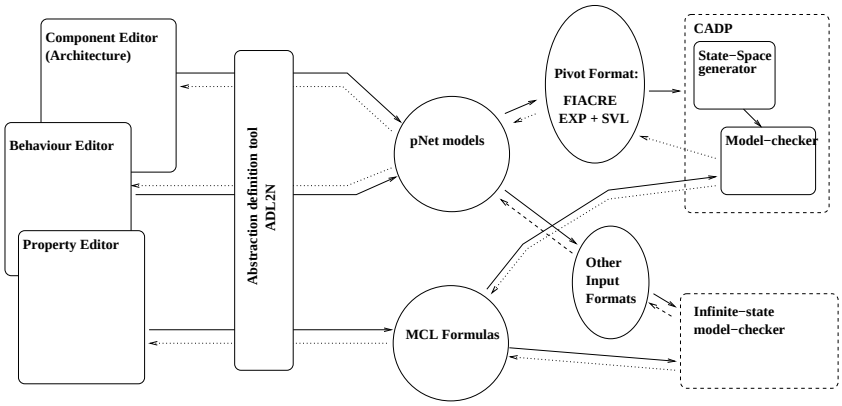


FIG. 5.3 – Foreseen architecture of VerCors verification backend(s)

New model-checking techniques Our team expertise does not include the model-checking methods, algorithms, or tools themselves. Instead we rather focus on the method and tools for building the program models, and using existing model-checkers. However, I will speak shortly here of two cases in which we had to make significant research work to adapt or extend new model-checking technologies to our needs.

The first one is about verification of **unbounded fifo channels**. There have been a number of works in the area of infinite systems model-checking, most of them addressing the search for (semi-) decidable sublogics, for modeling unbounded channels, processes with counters endowed with various arithmetic operators, or infinite structures of processes. But there have been very few realization ; the LASH library [89] is one of them, implementing data structures representing, among others, systems of finite state machines communicating over unbounded FIFO channels [22]. We could not use directly the LASH implementation, on one side because we wanted a java implementation for future integration in VerCors, on the other hand because the LASH implementation is quite general and we needed specific adaptation of the algorithm to have a proper control on the search strategies, and to be able to limit the search space using information specific to our model. We implemented a prototype providing a formalism for defining such machines, and an adaptation of the QDD algorithm. This specific "infinite state" domain is important for ProActive and GCM-like applications, where request queues are by default unbounded fifos. Our first results show that the approach is feasible for toy examples, but the practical complexity quite high. A workable implementation

would require 1) a thorough work on the efficiency of the data representation and of the algorithm 2) extension to non-fifo cases where the message selection policy is expressed in some regular manner, and 3) a way to integrate this algorithm (both in theory and in practice) with other finite-state and infinite-state tools.

The second case is about **distributed and hierarchical model-checking**. We have been using for a while the "Distributor" engine of the CADP toolset, which provides distributed state-space generation for applications expressed in a number of different CADP input formalisms. Distributor [49] uses many similar instances on as many different nodes of a cluster, each instance managing a predefined subset of the generated states, and communicating with others by message passing. The generated state space is stored in RAM during generation, so the global state space can be as large as the total RAM of the cluster... The current bottleneck of this method is that other tools of CADP (the bisimulation minimizer, or the model-checker itself) are not distributed, so the generated state-space has to be assembled in a single file (in bcg format) before applying these tools. We have built a tool for building "verification workflows" and running them on grid or cloud infrastructures. We used this in our group-communication case-study [R-10], and you will find figures in Chapter 7. The results are quite good for scaling up in term of the size of models (up to 10^{12} states explicitly stored), and make possible the combination of many techniques to reduce the state explosion, including hierarchical hiding and minimization, partial-order reduction, usage of process contexts, etc. In the current state we are severely lacking of tool support : 1) for generating the various parts of the pivot format for the different parts of the workflow tasks (Fiacre, EXP and SVL formats), 2) for automatically adapting the model generation to the set of formulas we want to prove, 3) for automatically deploying and monitoring the verification tasks on modern (elastic) computing infrastructures, and 4) for debugging purposes, and in particular for lifting back the model-checker diagnoses to the level of the user specification formalisms.

Perspectives The VerCors platform is already a powerful prototype for demonstrating our methods on large examples. But it needs a number of additional tools before being usable in a convenient way, and, more important in our perspective, to be usable by non-specialists. The missing parts are essentially :

- Input formalisms and editing tools for expressing the behaviour of basic objects (automata), and for defining the properties required (logic formulas). Whenever possible, we do not want to reinvent new languages, and we would rather prefer to use some existing formalism already known by developers. For automata, this could be some form of state-charts or activity diagrams. For Logic formulas, this is more difficult, but MCL (Model Checking Language [70]) could be a good candidate.
- Automatic tools fully implementing the model-generation procedures as described in Figure 5.3. These tools will include an extension of the abstraction operations implemented in the ADL2N tool, helping developers to define proper and consistent abstractions. They will consistently apply these abstractions to all the elements of the input formalisms, including the logic language. And they will support the rendering of debugging information directly in the input formalisms, providing the developer with readable diagnoses.

5.2 Paper from *FMCO Symposium, Sep. 2008*

Specification and Verification for Grid Component-Based Applications: From Models to Tools

Antonio Cansado and Eric Madelaine

INRIA – CNRS – I3S – Université de Nice Sophia-Antipolis
2004 Route des Lucioles, Sophia Antipolis - France
{acansado,madelain}@sophia.inria.fr

Abstract. Computer Grids offer large-scale infrastructures for computer intensive applications, as well as for new service-oriented paradigms. Programming such applications brings a number of difficulties due to asynchrony and dynamicity, and require specific verification methods. We define a behavioural model called pNets for describing the semantics of distributed component systems. pNets (for parameterized networks of synchronised automatas) are hierarchical assemblies of labelled transition systems, with data parameters expressing both value-passing and parameterized topology. We use pNets for building models for Fractal (hierarchical) and GCM (distributed) components. We present the VerCors platform, that implements these model generation procedures, but also abstraction mechanisms and connections with the model-checking engines of the CADP toolset.

1 Introduction

Software components [1] are the de facto standard in many information technology industries. Component-based frameworks and languages are seen as the natural successors of object-oriented languages for obtaining applications which are more modular, composable and reusable. Many solutions have been proposed during the past 10 years, with EJB being certainly the most well-known and used one. However, these promises are often considered from a software engineering perspective and are at best only empirically verified. We want to build development methods and environments that allow application designers to specify the external behaviour of software components in a black-box fashion, assemble them to build bigger components while guaranteeing that the parts will behave smoothly together, and check that such an assembly implements the overall behaviour expected by the user requirements. Beyond interoperability between components constituting large modern systems, e.g. in grid computing applications, or in large scale distributed software services, raise additional problems. In particular distributed and asynchronous components require more complex behaviour models, and the complexity of the analysis is higher. The analysis of properties related with reconfiguration and dynamicity brings new aspects to check, e.g. defining evolving systems, or checking substitutability.

Among the existing component models, *Fractal* [2] provides the following crucial features: the explicit definition of provided/required interfaces for expressing dependencies between components; a hierarchical structure allowing to build components

F.S. de Boer et al. (Eds.): FMCO 2008, LNCS 5751, pp. 180–203, 2009.

by composition of smaller components; and the definition of non-functional features through specific interfaces, providing a clear separation of concerns between functional and non-functional aspects. The *Grid Component Model (GCM)* [3], extends Fractal by addressing large scale distributed aspects of components, providing structures for asynchronous method calls with implicit futures[1], and NxM communication mechanisms. Both Fractal and GCM models provide means to specify and implement management and reconfiguration operations.

The objective of our work is to provide tools to the programmer of distributed components systems in order to verify the correct behaviour of programs. We require those tools to be intuitive and user-friendly to be usable by non-experts of formal methods. To this end we build an analysis toolset, including graphical editors for defining the architecture and the behaviour of components, and state-of-the-art model-checking tools. At the heart of this platform lie the behaviour semantics of our component systems, and the model generation tools that are the subject of this article. In this context the choice of the behavioural model is crucial: it has to be compact, expressive enough to represent the behavioural semantics, but not too much, that could prevent us to map the models to the input formats of automatic verification tools. Some recent approches, for example π-ADL [4], are using formalisms based on the π-calculus, others, like μ-CRL [5] or STS [6] use algebraic descriptions of data domains. In both cases, such foundations give them powerful primitives for describing dynamic or mobile architectures, but also strong limitations for using automatic verification.

Most established approaches, on the other side, are using intermediate formats with data, that can be unfolded to finite-state structures. This is the case e.g. for the CADP toolbox [7], or for the SPIN model-checker and its specification language PROMELA, whose data values are instantiated (on bound domains) by the state exploration engines.

Our choice is to use an intermediate approach with a compositional semantic model including data called pNets [8]. It is different from previous approaches in the sense that we want a low-level model able to express various mechanisms for distributed systems, and that we do not limit ourselves to finite systems: we shall be able to define mappings to various classes of systems, finite or not. At the same time, the structure of our parameterized model is closer to the programming language or the specification language structure. Consequently, parameterized models are more compact, and easier to produce, than classical internal models. Typically, our pNets model is lower level than Lotos and Promela, but more flexible for expressing different synchronisation mechanisms. On the other hand, it has no recursive constructs, in order to better control the finiteness of encodings.

The second half of this work is a set of software tools called VerCors [9] for specifying and verifying GCM component systems. In the middle term, it will include both a textual and a graphical specification languages, unifying the architectural and the behavioural description of components [10]. It provides tools for defining abstractions of the system, and for computing their behaviour model in term of pNets. Finally it

[1] This is in contrast with languages like MultiLisp or Creol, where futures are explicit in the code. Having implicit futures in GCM/ProActive allows us to automatically provide optimal asynchrony.

has bridges with the CADP verification toolset, allowing efficient (explicit) state-space construction, and model-checking.

In the next section we describe the context of this work, namely the formalisms and models that we use for hierarchical distributed components: Fractal and GCM, and the communication mechanisms of the GCM implementation ProActive. In section 3 we recall the definitions of the parameterized networks of synchronised automatas (pNets), and we give the definition of the behavioural semantics of distributed components, starting with active objects, then modelling hierarchical components, Fractal components, and finishing with the specific features of GCM components, including multicast and gathercast interfaces, and first-class futures. In section 4, we describe the VerCors specification and verification platform, with a glimpse at its architecture, a description of the graphical editors, of the model generation tool, and some results obtained with the platform.

2 Context: Asynchronous Component Model, Active Objects, Grids

2.1 ASP and Active Objects

The ASP calculus [11] is a distributed object calculus with futures featuring:

- asynchronous communications: by a request-reply mechanism,
- futures, that are promised replies of remote method invocations,
- sequential execution within each process: each object is manipulated by a single thread of control,
- imperative objects: each object has a state.

An essential design decision is the absence of sharing: objects live in disjoint activities. An activity is a set of objects managed by a unique process and a unique active object. Active objects are accessible through global/distant references. They communicate through asynchronous method calls with futures. A future is a global reference representing a result not yet computed. The main result consists in a confluence property and its application to the identification of a set of programs behaving deterministically. This property can be summarized as follows: future updates can occur at any time; execution is only characterized by the order of requests; programs communicating over trees are deterministic.

From the proposed framework, we have shown a path that can lead to a component calculus [12]. It demonstrates how we can go from asynchronous distributed objects to asynchronous distributed components, including collective remote method invocations (group communications), while retaining determinism.

The impact of this work on the development of the ProActive library on one hand, and on the building of the behavioural semantics on the other hand, is probably one of our strongest achievements.

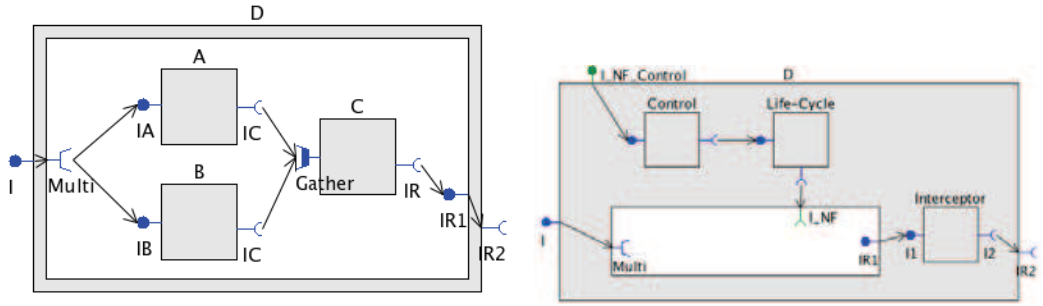
2.2 Fractal and GCM

Fractal [2] is a flexible and extensible component model. Its main features are: a hierarchical structure, in which everything can be built from components (including bindings and membranes), a generic description of non-functional concerns (e.g. life-cycle,

binding, attribute management) through specific control interfaces, a strong separation of concerns between functional and non-functional aspects, a well-defined architecture description language (ADL), and several implementations [13, 14].

The Grid Component Model (GCM) [3] is a novel component model that has been defined by the European Network of Excellence CoreGrid and implemented by the EU project GridCOMP. The GCM is based on Fractal, and extends it to address Grid concerns.

Grids consider thousands of computers all over the world; programming Grids involve dealing with latency in communications between computing nodes, and optimizing whenever possible the parallelism of the computation. For that, GCM extends Fractal using asynchronous method calls. Grid applications usually have numerous similar components, so the GCM defines collective interfaces which ease design and implementation of such parallel components by providing synchronisation and distribution capacities. There are two kinds of collective interfaces in the GCM: multicast (client) and gathercast (server).



(a) Content of a composite component (b) Membrane of a component

Fig. 1. GCM components

One to N and N to one interfaces. Typically a multicast interface (such as the interface `Multi` in Fig. 1(a)) is bound to the service interfaces of a number of parallel components, and a method call toward this interface is distributed, as well as its parameters, to several or all of them. GCM provides various policies for the request parameters, that can be broadcast, or scattered, or distributed in a round-robin fashion; additional policies can be specified by the user. The computation on the remote components will eventually terminate and send back, asynchronously, their results; Then the results of the invocations have to be assembled back with different possible policies (gather the results in a list, return the sum of the results, compute the maximum, or just pick the first that arrives and discard others...).

Symmetrically, gathercast interfaces (e.g. `Gather` in Figure 1(a)) are bound to a number of client components, and various synchronisation policies are provided. This corresponds to synchronisation barriers in message-based parallel programming, though here you may also have to specify how you redistribute the result on the client interfaces.

This treatment of collective communications provides a clear separation of concern between the programming of each component, and the management of the application topology: within a component code, method calls are addressed simply to the component local interfaces. The management of bindings of clients (on a gathercast interface) or services (on a multicast interface) is separated from the functional code.

Membranes and Non-functional interfaces. The component's non-functional (NF) aspects are handled by the component's membrane. The membrane is structured as a component system defining so-called *NF components*. Moreover, the GCM specifies interfaces for the autonomic management and adaptation of components. The membrane is also in charge of controlling the interaction between the component's content and the environment: the membrane decides how requests entering or leaving the component are to be treated.

The simplest binding one can define in a membrane is a binding from an external interface to an internal interface (e.g server interface I to internal interface `Multi` in Figure 1(b)): requests will simply be forwarded to a subcomponent server interface. But a NF component called `Interceptor` can be inserted between an external and an internal functional interface that will perform some non-functional processing (e.g. encrypting, logging, etc); an example is the `Interceptor` component between interfaces `IR1` and `IR2` in Fig. 1(b)).

More complex NF components can be used for introspection, reconfiguration, or autonomic management. Those will typically lie between the external and internal NF interfaces of the composite component.

Architecture. The Architecture Description Language (ADL) of both Fractal and the GCM is an XML-based format, that contains both the structural definition of the system components (subcomponents, interfaces and bindings), and some deployment concerns. Deployment relies on *virtual nodes* that are an abstraction of the physical infrastructure on which the application will be deployed. The ADL only refers to an abstract architecture, and the mapping between the abstract architecture and a real one is given separately as a deployment descriptor.

The Fractal/GCM ADL descriptions are static. Dynamicity of component applications, and the ability to reconfigure them, is gained through specific operations of their APIs. Several aspects of GCM, including its ADL, API, deployment description, application resources description, are now standardized by the European Telecommunication Standards Institute ETSI [15].

2.3 A GCM Reference Implementation: GCM/ProActive

The GCM reference implementation is based on ProActive [16], an Open Source middleware implementing the ASP calculus. In this implementation, an active object is used to implement each primitive component and each composite membrane. Although composite components do not have functional code themselves, they have a membrane that encapsulates controllers, and dispatches functional calls to inner subcomponents. As a consequence, this implementation also inherits some constraints and properties w.r.t. the programming model:

- components communicate through asynchronous method calls with transparent futures (place-holders for promised replies): a method call on a server interface adds a request in the server's *request queue*;
- communication semantics use a "rendez-vous" ensuring the causal ordering of communications;

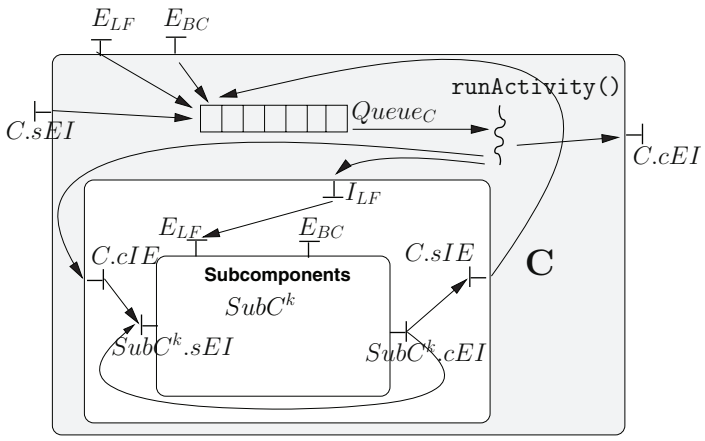


Fig. 2. *ProActive* composite component

– synchronisation between components is ensured with a data-flow synchronisation called *wait-by-necessity*: futures are first order objects that can be forwarded to any component in a non-blocking manner, execution is only blocked if the concrete value of the result is needed (accessed), while the result is still unavailable;
– there is no shared memory between components, and a single thread is available for each component.

Each primitive component is associated with an active object written by the programmer. Some methods of this active object are exported as the methods of the component's interfaces. The active object managing a composite is generic and provided by the GCM/ProActive platform; it forwards the functional requests it receives to its subcomponents. Primitive component functionalities are addressed by the encapsulated active object. For primitive components, it is possible to define the order in which requests are served by writing a specific method called `runActivity()`; we call this the service policy. If no `runActivity()` is given, a default one implements a FIFO policy (excepted for non-functional requests, see below). Composite components always use a FIFO policy. Note that futures create some kinds of implicit return channels, which are only used to return one value to a component that might need it.

Life-Cycle of GCM/ProActive Components. GCM/ProActive implements the membrane of a composite as an active object, thus it contains a unique request queue and a single service thread. The requests to its external server interfaces (including control requests) and from its internal client interfaces are dropped to its request queue. A graphical view of a composite is shown in Fig. 2.

Like in Fractal, when a component is stopped, only control requests are served. A component is started by invoking the non-functional request: `start()`. Because threads are non-interruptible in Java, a component necessarily finishes the request it is treating before being stopped. If a `runActivity()` method is specified by the programmer, the stop signal must be taken into account in this method.

Note that a *stopped* component will not emit functional calls on its required interfaces, even if its subcomponents are active and send requests to its internal interfaces.

2.4 Example

We will use the example in Fig. 3 to illustrate the various aspects of this paper. It is formed from one composite component B and three primitive components A, C, D. Component B has a number of subcomponents, and requests on its server interface S are dispatched to them through the multicast interface MC. Component D has two server interfaces W and R, and is supposed to host some shared resource (e.g. a database); its role in the example is to show the possible race-conditions or deadlocks that could arise, e.g, if a request on interface W has a side effect on the shared resource. Component A plays the client role, and will send requests to B, creating futures containing their promised responses, and transmitting these futures as parameters to requests to C. Component B also has two non-functional interfaces NF1 and NF2 that may be used e.g. to reconfigure its content.

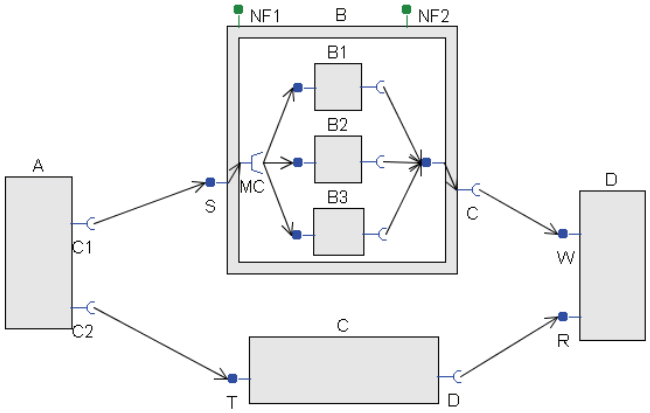


Fig. 3. Running example

3 Semantic Model

In this section, we recall the main definitions of the *parameterized Networks of synchronised automatas* (pNets, [8]). We use pNets as a general low level behaviour model for encoding different variants of our languages or component models. We start with the formal definitions of the model. Then we use pNets to define the behavioural semantics of two basic and important formalisms in the domain of distributed components: the ProActive "Active Objects" on one hand, and Fractal hierarchical components on the other hand (both examples are excerpts from [8]). Finally, we give an encoding for GCM components, including the management of request queues in primitives and composite components, and the encoding of future proxies, in presence of first class futures.

3.1 Parameterized Networks of Synchronised Automata (pNets)

The following definitions are taken from [8]. We start with classical labelled transition systems and structure them using synchronisation networks. Then we extend these definitions to include parameters, both as arguments in communication and in state definitions (à la "value-passing CCS"), and in synchronisation operators, obtaining a model powerful enough to describe parameterized and dynamic topologies.

We model the behaviour of a process as a Labelled Transition System (**LTS**) in a classical way [17]. The LTS transitions encode the actions that a process can perform in a given state.

Definition 1. LTS. *A labelled transition system is a tuple* $\langle S, s_0, L, \rightarrow \rangle$ *where* S *(possibly infinite) is the set of states,* $s_0 \in S$ *is the initial state,* L *is the set of labels,* $\rightarrow$ *is the set of transitions :* $\rightarrow \subseteq S\,xLxS$. *We write* $s \xrightarrow{\alpha} s'$ *for* $(s, \alpha, s') \in \rightarrow$.

We define **Nets** in a form inspired by the *synchronisation vectors* of Arnold and Nivat [18], that we use to synchronise a (potentially infinite) number of processes.

In the following definitions, we frequently use indexed vectors: we note $\tilde{x}_I$ the vector $\langle ..., x_i, ... \rangle$ with $i \in I$, where I is a countable set.

Definition 2. Network of LTSs.[2] *Let Act be an action set. A* **Net** *is a tuple* $\langle A_G, J, \tilde{O}_J, \overrightarrow{V} \rangle$ *where* $A_G \subseteq Act$ *is a set of global actions,* J *is a countable set of argument indexes, each index* $j \in J$ *is called a* hole *and is associated with a* sort $O_j \subset Act$. $\overrightarrow{V} = \{\overrightarrow{v}\}$ *is a set of synchronisation vectors of the form:* $\overrightarrow{v} = \langle a_g.\tilde{\alpha}_I \rangle$ *where* $a_g \in A_G$, $I \subseteq J \wedge \forall i \in I, \alpha_i \in O_i$

Fig. 4 gives a naive representation of the Net representing component B, with four subcomponents. Here the semantics has been configured so that `call` requests are going through a MC policy component, and are made visible (to the next level) as "`?call(m,args)`" for requests received by `B`, and "`B[i].call(m,args)`" for the requests dispatched to the respective `B[i]`. As an example, the second synchronisation vector in $\overrightarrow{V}$ reads as: action "`!call(m,x1)`" of the first hole (here `MC`) can occur synchronised with action "`?call(m,x1)`" of `B1`, and the corresponding global action is "`B[1].call(m,x1)`". There should be one such vector for each possible value of `x1`.

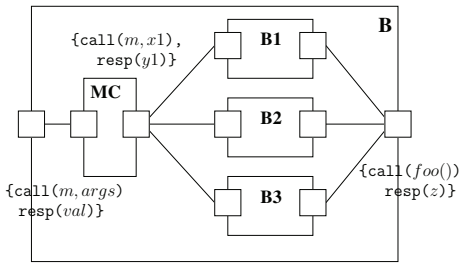
Note that the specific syntax (and meaning) of the actions is not important here: it depends on the specific formalism that has been translated into Nets. The synchronisation vectors are the only means that we use to express the synchronisation mechanisms. This way we can express traditional message passing (matching emission/reception), as well as other mechanisms like one to N synchronisation. In this first non parameterized version, we may need a infinite number of vectors to express the synchronisations occuring in a Net.

Definition 3. *A* **System** *is a tree-like structure whose nodes are* **Nets***, and leaves are* **LTSs**. *At each node a partial function maps holes to corresponding* **subsystems**. *A system is* **closed** *if all holes are mapped, and* **open** *otherwise.*

Definition 4. *The* **Sort** *of a system is the set of actions that can be observed from outside the system. It is determined by its top-level node, with:*

$$Sort(\langle S, s_0, L, \rightarrow \rangle) = L \qquad Sort(\langle A_G, J, \tilde{O}_J, \overrightarrow{V} \rangle) = A_G$$

[2] This definition is simpler than the one we gave in [8], from which we have removed the *transducer* element in the pNet structure. It is possible to obtain an expressiveness similar to pNets with transducers by adding an extra argument to each pNet, and specifying this "Controller" as an argument pLTS.



where *B-3-Net* = $\langle A_G, J, \tilde{O}_J, \overrightarrow{V} \rangle$ with:

A_G = {`?call(m,args), !resp(val), B1.call(m,x), ...`}

J = {`MC, B1, B2, B3`}

O_{MC} = {`?call(m,args), !resp(val), !call(m,x1), ...`}

$O_{B1} = O_{B2} = O_{B3}$ = {`?call(m,x), !resp(val), !call(foo()), ?resp(z)`}

$\overrightarrow{V}$ ={

```
⟨ ?call(m,args), ?MC.call(m,args), -, -, -⟩
⟨ B[1].call(m,x1), !B1.call(m,x1), ?call(m,x1), -, -⟩
⟨ B[2].call(m,x2), !B2.call(m,x2), -, ?call(m,x2), -⟩
... }
```

Fig. 4. Example of Net

Next we enrich the above definitions with parameters in the spirit of Symbolic Transition Graphs [19]. We start by giving the notion of parameterized actions. We leave unspecified here the constructors and operators of the action algebra, they will be defined together with the encoding of some specific formalism.

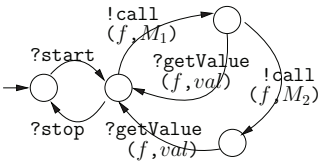
Definition 5. Parameterized Actions. *Let P be a set of names, $\mathcal{L}_{A,P}$ a term algebra built over P, including at least a distinguished sort Action, and a constant action τ. We call $v \in P$ a parameter, and $a \in \mathcal{L}_{A,P}$ a parameterized action, $\mathcal{B}_{A,P}$ the set of boolean expressions (guards) over $\mathcal{L}_{A,P}$.*

Definition 6. pLTS. *A parameterized LTS is a tuple $\langle P, S, s_0, L, \rightarrow \rangle$ where:*

- *P is a finite set of parameters, from which we construct the term algebra $\mathcal{L}_{A,P}$,*
- *S is a set of states; each state $s \in S$ is associated to a finite indexed set of free variables $fv(s) = \tilde{x}_{J_s} \subseteq P$,*
- *$s_0 \in S$ is the initial state,*
- *L is the set of labels, $\rightarrow$ the transition relation $\rightarrow \subset S \times L \times S$*
- *Labels have the form $l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} \rangle$ such that if $s \xrightarrow{l} s'$, then:*
  - *α is a parameterized action, expressing a combination of inputs $iv(\alpha) \subseteq P$ (defining new variables) and outputs $oe(\alpha)$ (using action expressions),*
  - *$e_b \in \mathcal{B}_{A,P}$ is the* optional *guard,*
  - *the variables $\tilde{x}_{J_{s'}}$ are assigned during the transition by the* optional *expressions $\tilde{e}_{J_{s'}}$*

  with the constraints: $fv(oe(\alpha)) \subseteq iv(\alpha) \cup \tilde{x}_{J_s}$ and $fv(e_b) \cup fv(\tilde{e}_{J_{s'}}) \subseteq iv(\alpha) \cup \tilde{x}_{J_s} \cup \tilde{x}_{J_{s'}}$.

Example: Fig. 5 represents a possible behaviour of the body of component A from our example. The action alphabet used here reflects the active object communication schema: each remote request sent by the body has the form "`!call(`$f, \mathcal{M}(\tilde{arg})$`)`", where $\mathcal{M}$ is the method name, eventually with parameters $\tilde{arg}$, and f is the identifier of the future proxy instance. Thus in this example, the action expressions are built from variables f and *val*, from the constants M_1 and M_2, and from the binary action constructors `call` and `getValue`. These actions allow the component to perform a remote method call, and



$A\text{-}LTS = \langle P, S, s_0, L, \rightarrow \rangle$
with:

$P = \{f, val\}$

$S = \{s_i\}$, `i ∈ [0:3]`

$L = \{$ `?start,?stop,!call(`f`,`M_1`),!call(`f`,`M_2`),?getValue(`f`,`val`)` $\}$

$\rightarrow$ such that:

s_0 : `?start` $\rightarrow s_1$,
s_1 : `?stop` $\rightarrow s_0$,
s_1 : `!call(`f`,`$M1$`)` $\rightarrow s_2$,
s_2 : `?getValue(`f`,`val`)` $\rightarrow s_1$
s_3 : `!call(`f`,`$M2$`)` $\rightarrow s_3$
s_4 : `?getValue(`f`,`val`)` $\rightarrow s_1$

Fig. 5. Behavioural model of component A

access the return value resp.; more details on how the component communicates with its environment are given later in Fig. 7.

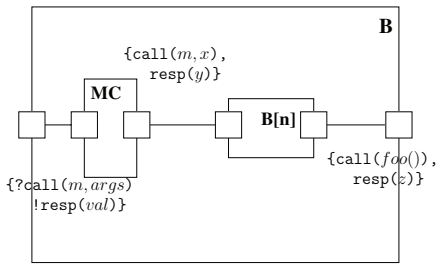
Now, we define pNets as Nets where the holes can be indexed by a parameter, to represent (potentially unbounded) families of similar arguments.

Definition 7. *A* **pNet** *is a tuple* $\langle P, pA_G, J, \tilde{p}_J, \tilde{O}_J, \overrightarrow{V} \rangle$ *where: P is a set of parameters,* $pA_G \subset \mathcal{L}_{A,P}$ *is its set of (parameterized) external actions, J is a finite set of holes, each hole j being associated with (at most) a parameter* $p_j \in P$ *and with a sort* $O_j \subset \mathcal{L}_{A,P}$*.* $\overrightarrow{V} = \{\overrightarrow{v}\}$ *is a set of synchronisation vectors of the form:* $\overrightarrow{v} = \langle a_g, \{\alpha_t\}_{i \in I, t \in B_i} \rangle$ *such that:* $I \subseteq J \wedge B_i \subseteq \mathcal{D}om(p_i) \wedge \alpha_i \in O_i \wedge fv(\alpha_i) \subseteq P$

Explanations: Each hole in the pNet has a parameter p_j, expressing that this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation of its parameter (we could have, without changing the expressiveness, several parameters per hole). In other words, the parameterized holes express *parameterized topologies* of processes synchronised by a given Net. Each parameterized synchronisation vector in the pNet expresses a synchronisation between some instances ($\{t\}_{t \in B_i}$) of some of the pNet holes ($I \subseteq J$). The hole parameters being part of the variables of the action algebra, they can be used in communication and synchronisation between the processes.

Fig. 6 is the parameterized version of the pNets for component B, in which the second hole (`B`) has a parameter `n`. The second synchronisation vector in the examples synchronises one (parameterized) action of the first hole `MC`, with an action (`?call(m,x)`) of the n^{th} instance of `B`. The comparison with the instantiated version in Fig. 4 shows clearly the benefits of parameterization, in term of compactness, and of generality. Note that this is still a very simplified and naive version of the pNet for B, the full semantics of GCM composite components will be given later.

A pNet by itself is stateless, but it has state variables that encode some notion of internal memory that can influence the synchronisation. pNets have the nice property that they can be easily represented graphically, e.g. using the Autograph editor [20].



where *B-param-Net* = $\langle P, pA_G, J, \tilde{p}_J, \tilde{O}_J, \overrightarrow{V} \rangle$ with:
P = {n, args, val, x}
pA_G = {?call(m,args), !resp(val), B[n].call(m,x), ...}
J = {MC, B}
p_{MC} = {}, p_B = {n}
O_{MC} = {?call(m,args), !resp(val), !call(m,x), ?resp(y)}
O_B = {?call(m,x), !resp(val), !call(foo()), !resp(z)}
$\overrightarrow{V}$ ={
⟨ ?call(m,args), ?call(m,args), - ⟩
⟨ B[n].call(m,x), !B(n).call(m,x), n&?call(m,x)⟩
... }

Fig. 6. Example of a pNet

Building hierarchical pNets. Once a pNet hierarchical system is built, you need operations to transform it, and, at least:

- a product operation for reducing a pNets hierarchy to a flat pLTS,
- a way of instantiating a parameterized pNet system with respect to a given domain for one or several of its parameters.

In [8], we gave the definition of pNets instantiation, and we defined the product operation only for fully instantiated systems. This is enough for instantiating a pNet system for some finite abstraction of the parameter domains, and building the global state-space of the system.

3.2 Model Generation for Active Objects

The first application of pNets that we have published was for defining the behavioural semantics of active objects of the ProActive library. In [21, 22] we presented a methodology for generating behavioural models for active objects (AOs), based on static analysis of the Java/ProActive code. The pNets model fits well in this context, and allows us to build compact models, with a natural relation to the code structure: we associate a hierarchical pNet to each active object of the application, and build a synchronisation network to represent the communication between them.

Fig. 7 illustrates the structure of the pNets expressing an asynchronous communication between two active objects. A method call to a remote activity goes through a proxy, that encodes the creation of the local future object, while the request goes to the remote request queue. Note that for each program point pp corresponding to a remote method call in the source code, a series of futures, indexed by a counter c, can be created. The request arguments include the references to the caller and callee objects, but also to the future. Later, the request may eventually be served, and its result value will be sent back and used to update the future value.

This method is composed of two steps: first the source code is analysed by classical compilation techniques, with a special attention to tracking references to remote objects in the code, and identifying remote method calls. This analysis produces a graph including the method call graph and some data-flow information. The second step consists in

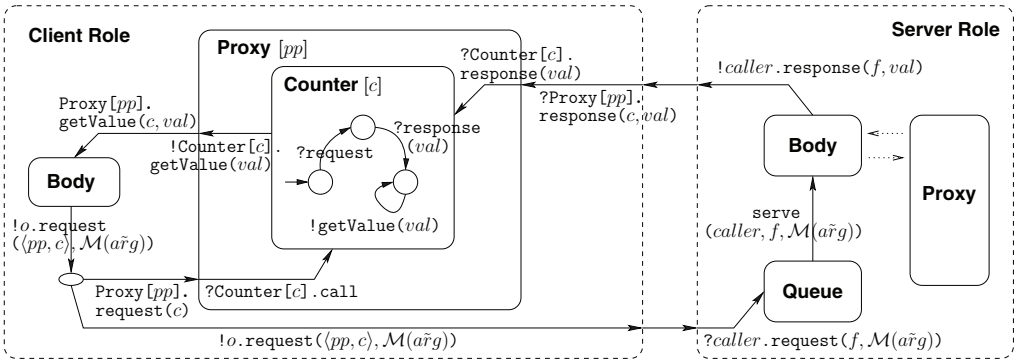


Fig. 7. Communication between two Active Objects

applying a set of structured operational semantics (SOS) rules to the graph, computing the states and transitions of the behavioural model.

The construction of the extended graphs by static analysis is technically difficult, and fundamentally imprecise. Imprecision comes from classical reasons (having only static information about variables, types, etc), but also from specific sources: it may not be decidable statically whether a variable references a local or a remote object. Furthermore, the middleware libraries include a lot of dynamic code generation, and the analysis would not be possible for Java code relying on introspection, classically used to manage some types of "dynamic topologies" in ProActive.

3.3 Model Generation for Hierarchical Components

Going from active objects to distributed and hierarchical components allows us to gain precision in the generated models. The most significant difference is that required interfaces are explicitly declared, and active objects are statically identified by components, so we always know whether a method call is local or remote. Moreover, the pNets's formalism expresses naturally the hierarchical structure of components, and will allow to scale up better, using compositional verification methods,

The pNet construction here may apply to any kind of hierarchical component model that features:

- Components with a set of interfaces and a content.
- Interfaces typed by a set of methods with their signature.
- Bindings between sibling subcomponents, or between a component and one of its subcomponent.
- Composite content composed of subcomponents, internal interfaces, and bindings.
- Empty content for primitive components.

We leave here undefined the code of a primitive component. It will depend on the framework, and will be used to generate a pLTS representing the primitive behaviour. We also leave undefined the data domains used for specifying indexes within the parameterized structure, and for building the arguments of the method calls.

From the information in a Component structure, it is straightforward to generate a pNet representing the communication between the interfaces and the subcomponents, from the following elements:

- the pNet has one hole for each (parametric) subcomponent;
- the global actions pA_G and hole sorts $\tilde{O}_J$ of the pNets are sets of actions of the form $[!|?]\, C_i.Itf.\mathcal{M}(\overrightarrow{arg})$ for invoking/serving a method $\mathcal{M}$ on the interface *Itf*.
- for each binding, and for each method in the signature of the source interface of the binding, it has two parameterized synchronisation vectors, one for sending the request, and one for receiving the response.

3.4 Hierarchical Components + Management Interfaces = Fractal

In the Fractal model, and in Fractal implementations, the ADL describes a static view of the architecture (used to build the initial component system through a *component factory*), and non-functional (NF) interfaces are used to control dynamically the evolution of the system. In this section we consider the core of the Fractal model, containing the hierarchical structure from the previous section, plus the basic non-functional interfaces and controllers, namely the Life-Cycle Controller (LF) and the Binding Controller (BC). We defined the behavioural semantics of Fractal applications in terms of pNets, giving the overall structure of the pNets encoding primitive and composite components, and the pLTS defining the LF and BC controllers.

A life controller pLTS (see Fig. 8) is attached to each component. Control actions (`start`/`stop`) are synchronised with the parent component and with all of its subcomponents. Status actions (`started`/`stopped`) are synchronised with the component's functional behaviour and with the BC, because the BC may only allow rebinding of interfaces when stopped.

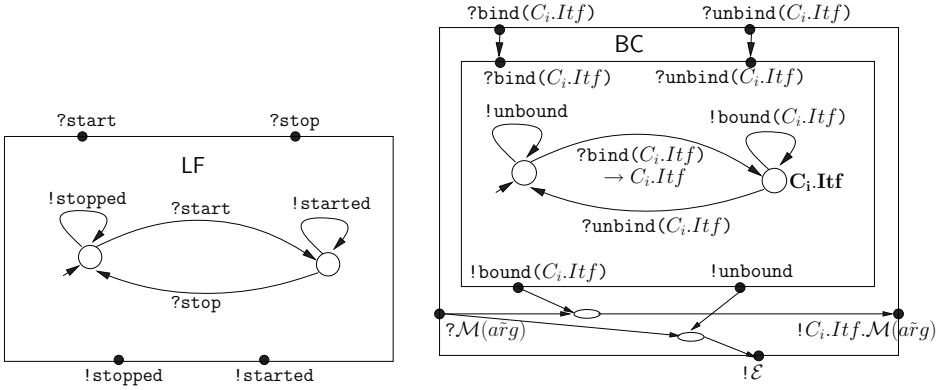


Fig. 8. pLTS of Fractal Life Cycle and Binding Controllers

A binding controller pLTS (see Fig. 8) is attached to each interface. Control actions (`bind`/`unbind`) are synchronised up to the higher level (Fractal defines a white-box definition for NF actions) and with the affected interface; status actions (`bound`/`unbound`) are used to allow method calls $\mathcal{M}(\tilde{arg})$, to forward the call to the appropriate bound interface and to signal errors. The latter is a distinguished action $\mathcal{E}(unbound, C, Itf)$, visible to the higher level of hierarchy, and triggered whenever a method call is performed over an unbound interface.

Fig. 9 sketches the structure of the synchronisation of a component with its subcomponents. In this drawing, the behaviour of subcomponents is represented by the box

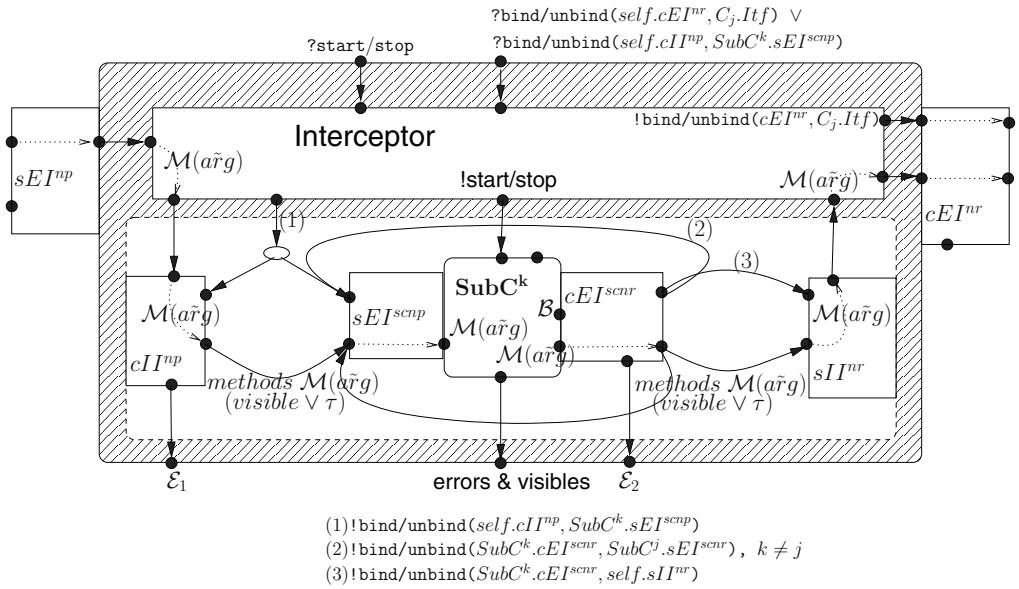


Fig. 9. Synchronisation pNet for a Fractal Composite Component

named *SubC*$^{\mathbf{k}}$. For each interface defined in the component's ADL description, a box encoding the behaviour of its internal (*cII* and *sII*) and external (*cEI* and *sEI*) views is incorporated. The dotted lines inside the boxes indicate a causality relation induced by the data flow through the box. Primitive components have a similar automaton without subcomponents and internal interfaces.

3.5 Model Generation for GCM

In Figure 10, we show the behavioural model of a GCM primitive component. There is a pLTS for dealing with the component's life-cycle (**LF**), and a pLTS for serving functional and non-functional requests (**Service**). The behavioural model for a composite component is an instance of the model of Figure 9, in which the interceptor itself is a primitive component.

Service implements the treatment of control requests. It interacts with the **LF** controller through the `!start` and `!stop` actions. The action `!start` fires the process representing the `runActivity()` method in the **Body**, and at the same time changes the LF state to "started". The `!stop` action is more complicated: it is sent by **Service** to the **Body**, but a running body may not be able to stop immediately upon reception of a stop request (because Java is non-interruptible). If the service policy of the component is the default FIFO, this `stop` request will be executed when all previous requests will be served. If the developer has specified his own `runActivity()` method, she/he has the responsibility for testing the presence of a `stop` request, and terminate the `runActivity()` method. At this point the `!stop` action will be transmitted to the **LF** controller, while the **Body** will be back in its initial state, ready for receiving a `!start` action.

The **Queue** pNet encodes an unbounded Fifo queue, containing requests composed by a method name and its arguments, and a selection mode (typically oldest or younguest request matching a predicate). It is always ready to perform any of the three actions numbered (1) to (3) in Fig. 10:

– (1) serve the first functional method obeying the selection mode;
– (2) serve a control method only at the head of the queue;
– (3) serve only control methods in FIFO order, bypassing the functional ones.

Depending on the state of the life-cycle controller, these actions may or may not synchronise with the Body and the Service pNets. This is encoded through the emission of the `!started` or `!stopped` actions by the LF pNet.

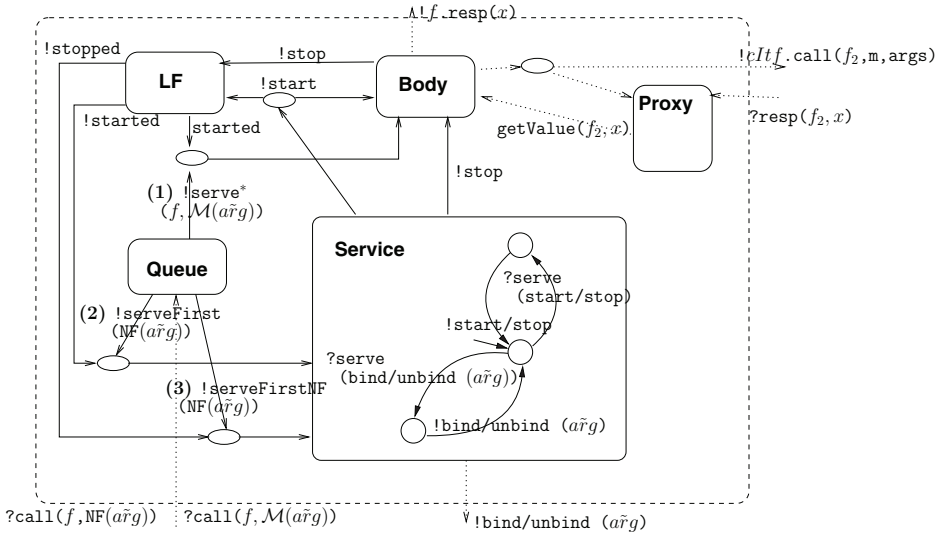


Fig. 10. Behavioural model of a primitive component

Modelling Collective Interfaces. Collective interfaces are responsible of distributing and gathering request calls and responses. Therefore, we provide a particular kind of proxy pLTS and N-ary synchronisation vectors encoding the control and data flow of these interfaces.

In Fig. 3, the multicast interface `MC` broadcasts request calls to all `B`'s subcomponents and gathers the results. We gave incomplete views of its pNet model, in Figures 4 and 6, and we show now its complete model in Figure 11. The proxy `Multicast(`f`)` pLTS is in charge of distributing the requests to all bound interfaces (in this case the server interfaces of `B`'s subcomponents). We use N-ary synchronisation vectors for broadcasting the call (`!call(args)`). This ensures that the call will be enqueued in every subcomponent at the same time. On the contrary, the response values of each component (`?resp(`*val*`)`) are sent back to the proxy individually and in any order. The proxy is in charge of gathering the result values in a vector. Later, when all results have arrived (guaranteed by the guard `[rep==N]`), it allows the component to access the result (`!getValue(`f`,`x`)`).

Modelling First-Call Futures. In Fig. 7 we depicted a simple proxy structure for ProActive futures. In GCM, futures can be transmitted in the parameters of a method call, or in the return value of a method call. In a naive approach, this requires knowing statically the flow of futures for each component because a future may have been created locally or by a third-party. This requires the analysis of the complete system. Instead, a better approach is to assign locally in each component an identifier f_{id} for

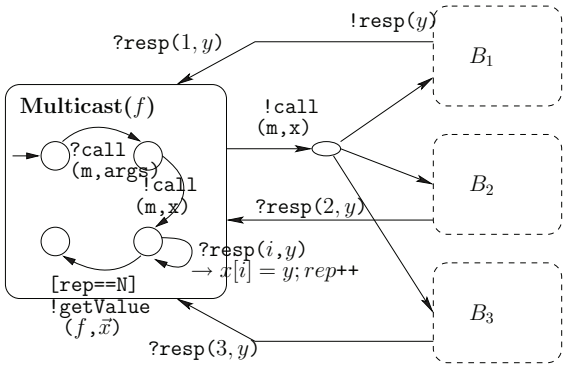


Fig. 11. Behavioural model of a multicast interface

each future, which permits the construction of behavioural models independently from the environment. Later, when the environment is known, the data-flow between components will determine which identifiers represent the same future object. At this point, these identifiers will be put in correspondence, and will be matched in the corresponding synchronisation vectors. This approach yields a compositional model.

In [23] we have shown the technical details of how to address different scenarios depending whether (i) a component transmits a locally created future; (ii) a component receives a future; and (iii) a component receives a future and retransmits it to a third-party. Here we define a new generic proxy that is able to deal with any combination of the 3 scenarios above. The proxy model has additional transitions w.r.t. the model presented in Figure 7 to allow futures to be transmitted. Figure 12 depicts this proxy[3].

When the local component is the creator of the future, the proxy starts by a transition `?call`. This allows the component to perform the remote remote call. In this case the proxy will wait for the `?response` transition to synchronise on the response value. Then there is a transition `!forward` for transmitting the future value to all components (if any) that may receive the future reference. Finally, the component body may access the content of the future through a `!getValue` transition.

Complementarily, if the local component did not create the future, the first transition of the proxy is a `?forward` which receives the value of the future. Afterward, the proxy behaves as in the previous case: it transmits the value to the remote components, and allows the component to access the future value.

Example: Sending a future created locally as a method call parameter. In Figure 12, the `Client` performs a method call $\mathcal{M}_1$ on `Server-A`, and creates a `Proxy(`f`)` for dealing with the result. Then the `Client` sends the future to a third activity (`Server-B`) in the parameter of the method $\mathcal{M}_2(f)$ (this call should eventually create another future f_2, but we have omitted it for simplicity).

[3] In this modelisation, we have an unbounded number of proxy instances, that live forever, and don't need to be terminated/destroyed. In the implementation, we may want to be more efficient: based on static analysis, the implementation can decide that some futures have a limited life-time, and that they can be destroyed or recycled at some point. Then we may want to prove correctness of such an optimisation.

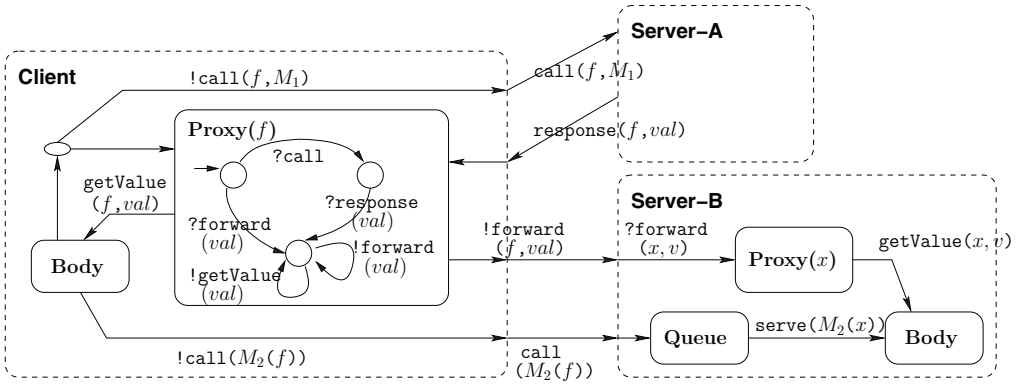


Fig. 12. Model for sending a future created locally as a method call parameter

From `Server-B`'s point of view, there is no way of knowing if a parameter is (or contains) a future, so every parameter in a method call must be considered as a potential future. `Server-B` includes, therefore, a proxy for dealing with the parameter x of the method call $\mathcal{M}_2$.

This example concludes the construction of pNets models for GCM components, incorporating non-functional controllers, request queues, future proxies, and NxM communication. In the current implementation, described in the next sections, the NxM communication and the proxies for first class futures are not yet supported.

4 VerCors: A Toolset for Specification and Verification

In this section, we report on the tool developments ongoing within our VerCors platform, implementing the behaviour model generation explained in the first half of this paper. We start with a description of the current and middle term functionalities of the platform, and we explain briefly the software tools used for the construction of the platform. Then we give more details on the graphical editors, on the model generation tool, and the model instantiation tools. Finally, we discuss some pragmatic aspects of various verification strategies for using the tools, and give some figures on typical case-studies.

4.1 Vercors Architecture

Fig. 13 sketches the architecture of VerCors. This toolset is available as free software, from our web site [9]. The platform has two goals: the verification of designs, and the generation of safe-by-construction code. In the following description of the VerCors modules, we shall indicate which functionalities are already available in the distribution (V0.2, spring 2009), and which are still under construction.

Front-End. VCE (for Vercors Component Editor) is our graphical component editor for designing components. It provides diagrams for defining the component architecture (see Section 4.3), and diagrams for defining the component behaviour (see Section 4.4). The latter is not yet available in V0.2. The Java Distributed Components specification language (JDC) is a textual language more expressive than our graphical diagrams, but is not yet implemented. It has been described in [10, 24].

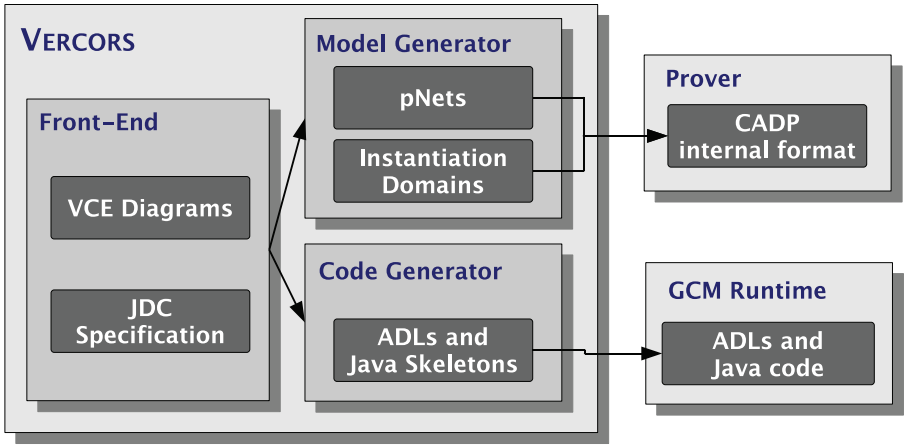


Fig. 13. The VerCors toolset

Model Generator. The model generator is the kernel of the platform. It is fed with specifications given by VCE diagrams or JDC specifications. It includes tools for data abstraction (from user-defined classes in JDC to Simple Types in pNets), tools for building the parameterized models from the specifications, and tools for manipulating and instantiating pNets (see section 4.5).

Code Generator. Another central part of the platform will be the code generator that is not (yet) currently developed. We will generate code capable of running under the standard GCM specification. It has an architecture definition based on the GCM ADL and Java code based on GCM / ProActive framework. The latter must be refined by the user by filling-in the business code.

External Tools. Externally to the platform, we interact with model-checking engines and with the GCM runtime. For now, VerCors uses the CADP toolset [25] for distributed state-space generation, hierarchical minimization, on-the-fly verification, and equivalence checking (strong/weak bisimulation). The connection with CADP is done through various textual input formats, that we generate from (fully instantiated) pNet models. A better approach would be to use a more generic and standardized intermediate format, like the FIACRE format [26], that would allow us to represent directly many (parameterized) constructs from the pNet model.

Verification is done by verifying regular μ-calculus formula encoding the user requirements. In the future, we would like to specify these properties within JDC, which would be subject to the same abstractions, and finally be translated into regular μ-calculus formula. We also plan to use other state-of-the-art provers, and in particular apply so-called "infinite system" provers to deal directly with certain types of parameterized systems.

4.2 Building Tools Using Eclipse Meta-modelling Framework

From a practical point of view, VCE consists of graphical editors for specifying the architecture and the behaviour of distributed components. It is built as an Eclipse plug-in based on EMF and GEF.

We use two similar meta-modelling frameworks, namely Topcased [27] and GMF. EMF plays the role of the *domain model* whereas Topcased and GMF provide graphical editors on top of the domain model. Unfortunately, Topcased is slowing down the development of their meta-modelling framework and future support is uncertain. Therefore, our early work on the architectural editor is generated by Topcased, but our more recent work on the behavioural editor is generated by GMF.

Model validation is based on OCL (Object Constraint Language) [28] rules that validate instances of the meta-model, and Java code that checks interface compatibility. There are a minimum set of invariants that every model must hold. Complementary, an additional set of rules cope with particular GCM implementations. All errors in the user models are reported in the Eclipse environment.

There is also compatibility with GCM ADL files. VCE is able to import and export GCM ADL files, though this is limited to functional components since there is no standard definition of NF components in the GCM ADL.

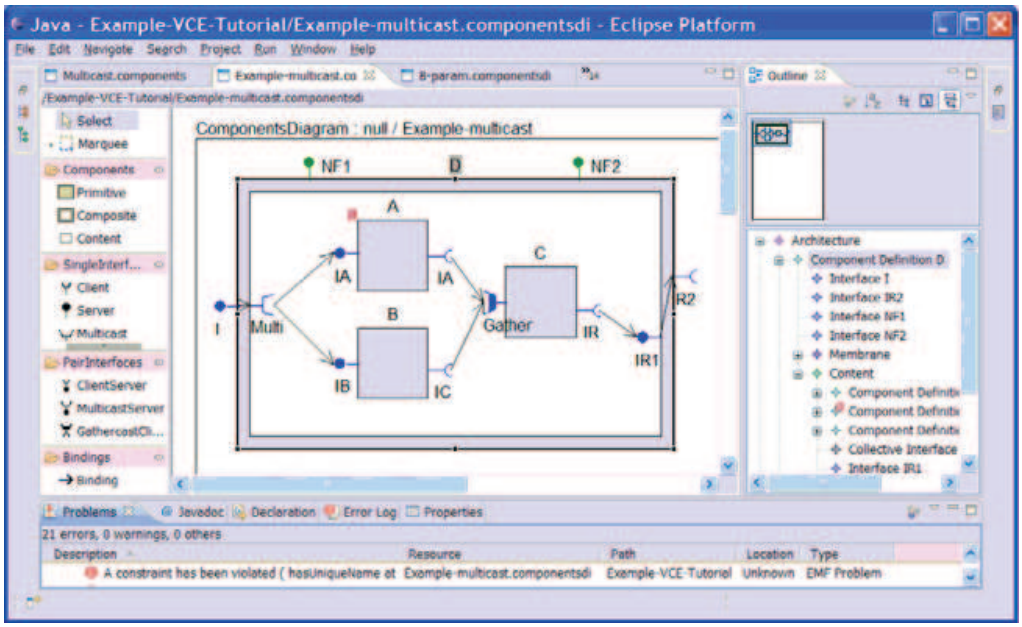


Fig. 14. Vercors Component Editor

4.3 Graphical Diagrams for Component Architecture

The kernel of the graphical language is a meta-model that reflects the GCM component structure. As these graphical constructions have already been used throughout this paper, we will only comment here on the main design choices that we have made.

At the top-level, the designer defines the root component that sets the services to be provided and required by the application to the environment. A component has a *content* that implements the business code, and a *membrane* that contains the non-functional code.

Components in the content are called *functional* components and those in the membrane are called *non-functional* (NF) components. The content is represented as a white

rectangle inside the component, and the membrane is the grey area that surrounds the content. Nevertheless, the content of primitive components is not depicted; therefore, primitive components are distinguished as grey rectangles. We colour blue the "usual" functional interfaces, and green the NF interfaces.

Interface icons are inspired by the ones used in UML component diagrams. Server interfaces are drawn as filled circles (e.g. interfaces `I`, `IA`, ... in Figure 14), and client interfaces as semi-circles (e.g. interfaces `IC`, `IR`, ...). GCM's *collective interfaces* are not defined in UML and hence we adopted our own icons. Figure 14 also shows the icons we provide for *multicast* and *gathercast* interfaces, labelled `Multi` and `Gather` respectively. In the example, the interface `Multi` broadcasts incoming requests to components `A` and `B`, and the interface `Gather` gathers and synchronises requests coming from interfaces `IC` of components `A` and `B` towards the component `C`.

4.4 Diagrams for Behaviour Specification

The diagrams for behaviour specification have been defined in [29], but the diagram editors are not yet available in the toolset. They are based on a variant of UML 2 State Machine diagrams, with a number of State Machines used to specify respectively: the component service policy, each service method and each local method, the interface policies, etc.

4.5 Model Generation

The role of the ADL2N tool is to:

- build an abstract version of the component system, in which the user-defined Java classes used for the parameter domains are abstracted by some Simple Types from the pNets library.
- use the behaviour semantics defined in sections 3.3 to 3.5 to build the pNet model for each piece of the system.

The first step of the model generation deals with data abstraction: data types in a JDC specification are standard, user-defined Java classes, but they must be mapped to Simple Types before generating the behavioural models and running the verification tools. The result is an abstract specification with the same structure than the initial ADL.

In practice the user of ADL2N uses a GUI to specify at the same time the methods that will be visible, the arguments that are significant for the proofs, and finite domains for these arguments. This is shown in Fig. 15. Here some tool guidance would be very helpful to reduce the amount of user input required, and to guarantee the coherency of the abstraction with the dataflow within the system. This kind of guidance is not yet available in the toolset.

Such an Abstract Specification will then be given as input to the model generator. This tool builds a model in terms of pNets, including all necessary controllers for non-functional and asynchronous capabilities of the components. The only missing part is the functional behaviour (Body) of primitive components for which ADL2N only defines their sorts.

The second usage of the abstraction module of ADL2N is to specify a *finite* abstraction of the parameters domains (from Simple Types to finite Simple Types), so that the

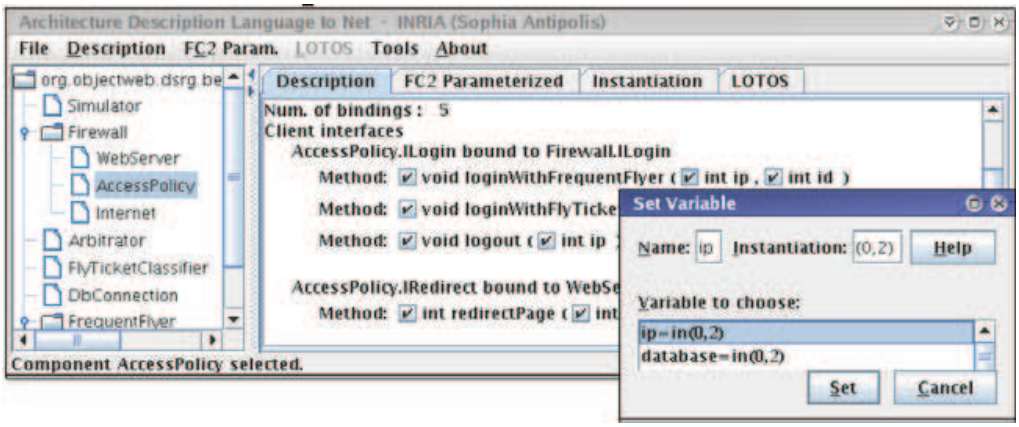


Fig. 15. Screenshot of ADL2N

final pNet system is finite, and suitable for analysis with finite-state model-checkers. In practice ADL2N produces two files, one file with the parameterized system, the other file with the definitions of the finite instantiations for the parameter domains.

pNets instantiations and export formats. The textual notation we use currently in the platform to encode pNets is called FC2 [30]. We provide two tools, FC2Instantiate and FC2Exp [31], that create finite instantiations of the models and transform the files into the input formats of CADP, namely BCG for transition systems, and Exp for synchronisation vectors [32].

4.6 Model-Checking: Engineering, Pragmatic Complexity

Having produced our models in a structured and hierarchical format allows us to use many pragmatic strategies to master as much as possible the state-space complexity of model-checking. The main tool is compositionality: as we use a bisimulation-based verification toolset, it is essential that each intermediate subsystem is reduced (by branching or weak minimization) before being synchronized with others. If we are careful to reduce as much as possible the visibility of actions, then state-space explosion can be contained (to some extent) within the model of composite components. Additionally, a number of advanced features of the CADP toolset can help us to fight state-explosion, and to scale up. Typically, we can build the state-space at each level of the hierarchy using the distributed state-space generation of CADP, including on-the-fly hiding and tau-reduction, but also behaviour generation constrained by the environment. Then the minimization has to take place on a single machine, because the bisimulation engine is not implemented in a distributed way. And the next cycle of construction can be distributed again... This way your state-space construction can scale up to any system in which the largest intermediate structure will be in the range of 10^8 states. The model-checker engine itself has an experimental version working in a distributed fashion.

Using this kind of strategy, we have done some middle-size case studies, including for example the Common Component Modeling Example (CoCoME, [33]). This is a system of 17 components structured in 5 levels of hierarchy, with more than 10 data parameters, and some broadcast communication. We have treated this case using the

Fractal model generation (3.4), with very small abstract domains for the variables (typically 2 or 3 values). The brute force state space for this would be approximately 2.10^8, while the biggest intermediate structure that we generate is lower than 10000 states. We have shown in [33] a number of properties and problems verified on this model.

Such models can be used to check the satisfiability of safety or liveness formulas in branching time logics, or to check the bisimulation equivalence with respect to an abstract specification. In practice, we want to provide non-expert users with simple "press button" verification functions. This is easy for some families of reachability properties, like correct termination of deployment, or occurrence of some predefined sets of error actions. Deadlock detection is also a popular "push button" function, but explaining to the user the reasons of a deadlock can be challenging; it often involves some "missed synchronisation", that may be difficult to show, especially in presence of abstraction and instantiation.

The type of properties we can check on our models are more versatile than in most approaches, because we do not only encode the usual functional interactions between the components, but also their reconfiguration operations. So we can prove properties of applications in which one would change bindings, or remove and update subcomponents, while the rest of the system keeps running. This kind of properties typically depends on the behaviour of the system parts, and is not a general property of the middleware.

5 Conclusion and Perspectives

In this paper we have presented the models and tools we have been implementing to assist the development of Grid component-based applications. The approach is based on the modelling of the component behaviour using parameterized networks of automata. In addition, we have presented tools that generate these models, and tools for the specification of the component system.

This paper makes a step forward towards the verification of Grid applications. It provides novel models for multicast interfaces and generic proxies for transmitting futures. Moreover, one of the strong original aspects of this work is the focus put on non-functional properties, and the results we provide on the interleaving between functional and non-functional concerns. Thus, the programmer should be able to prove the correct behaviour of his distributed component system in presence of evolution (or reconfiguration) of the system.

We are currently developing additional tools in the VerCors platform to support our methodology. This includes the front-ends for textual and graphical specification languages, a tool for helping the user to build correct abstractions, and tools for providing readable explanations of the provers diagnostics.

Finally, we have presented techniques to master state-space explosion. The key aspect is the use of compositionality to reduce the system at each level of hierarchy. Nevertheless, in some cases, particularly when queues are unbounded, state-space explosion is inevitable when using explicit-state model-checkers. Therefore, our latest work focuses on the development of an infinite-state model-checker that verifies automata endowed with unbounded FIFO queues.

References

[1] Szyperski, C.: Component Software, 2nd edn. Addison-Wesley, Reading (2002)

[2] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An open component model and its support in java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)

[3] CoreGRID, Programming Model Institute: Basic features of the grid component model (assessed). Technical report, CoreGRID, Programming Model Virtual Institute, Deliverable D.PM.04 (2006), http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf

[4] Oquendo, F.: π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. ACM Software Engineering Notes 26(3) (2004)

[5] Groote, J., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The Formal Specification Language mCRL2. In: Proc. Methods for Modelling Software Systems (2007)

[6] Poizat, P., Royer, J.-C., Salaün, G.: Bounded Analysis and Decomposition for Behavioural Descriptions of Components. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 33–47. Springer, Heidelberg (2006)

[7] Garavel, H., Lang, F., Mateescu, R.: An overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter 4, 13–24 (2002)

[8] Barros, T., Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed Fractal components. Annals of Telecommunications 64(1-2) (January 2009); also Research Report INRIA RR-6491.

[9] OASIS team: VerCors: a Specification and Verification Platform for Distributed Applications (2007-2009), http://www-sop.inria.fr/oasis/index.php?page=vercors

[10] Cansado, A., Henrio, L., Madelaine, E., Valenzuela, P.: Unifying architectural and behavioural specifications of distributed components. In: International Workshop on Formal Aspects of Component Software (FACS 2008), Malaga, Electronic Notes in Theoretical Computer Science (ENTCS) (September 2008)

[11] Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer, Heidelberg (2005)

[12] Caromel, D., Henrio, L.: Asynchonous distributed components: Concurrency and determinacy. In: Proceedings of the IFIP International Conference on Theoretical Computer Science 2006 (IFIP TCS 2006), Santiago, Chile, August 2006. Springer Science (2006); 19th IFIP World Computer Congress

[13] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An open component model and its support in java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)

[14] Seinturier, L., Pessemier, N., Coupaye, T.: AOKell: an Aspect-Oriented Implementation of the Fractal Specifications (2005), http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html

[15] European Telecommunication Standards Institute, http://portal.etsi.org

[16] Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on grids and P2P systems. Computational Methods in Science and Technology 12(1), 69–77 (2006)

[17] Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)

[18] Arnold, A.: Finite transition systems. Semantics of communicating sytems. Prentice-Hall, Englewood Cliffs (1994)

[19] Lin, H.: Symbolic transition graph with assignment. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119. Springer, Heidelberg (1996)

[20] Madelaine, E.: Verification tools from the CONCUR project. EATCS Bull. 47 (1992)
[21] Barros, T., Boulifa, R., Madelaine, E.: Parameterized models for distributed java objects. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004, Madrid. LNCS, vol. 3235, pp. 43–60. Springer, Heidelberg (2004)
[22] Boulifa, R.: Génération de modèles comportementaux des applications réparties. PhD thesis, University of Nice - Sophia Antipolis – UFR Sciences (December 2004)
[23] Cansado, A., Henrio, L., Madelaine, E.: Transparent first-class futures and distributed component. In: International Workshop on Formal Aspects of Component Software (FACS 2008), Malaga, Electronic Notes in Theoretical Computer Science, ENTCS (September 2008)
[24] Cansado, A.: Formal Specification and Verification of Distributed Component Systems. PhD thesis, Université de Nice - Sophia Antipolis – UFR Sciences (December 2008)
[25] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: CAV (2007)
[26] Berthomieu, B., Bodeveix, J.P., Filali, M., Garavel, H., Lang, F., Peres, F., Saad, R., Stoecker, J., Fran, C.V.: The Syntax and Semantics of FIACRE V2.0. Technical report (Feburary 2009)
[27] Pontisso, N., Chemouil, D.: Topcased combining formal methods with model-driven engineering. In: ASE, pp. 359–360. IEEE Computer Society, Los Alamitos (2006)
[28] Object Management Group: UML 2.0 Object Constraint Language (OCL) Specification. formal/03-10-14 edn, version 2.0 (2003)
[29] Ahumada, S., Apvrille, L., Barros, T., Cansado, A., Madelaine, E., Salageanu, E.: Specifying Fractal and GCM Components With UML. In: Proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), Iquique, Chile, Nov 2007, IEEE, Los Alamitos (2007)
[30] Ressouche, A., de Simone, R., Bouali, A., Roy, V.: The FC2Tool user manuel (1994), http://www-sop.inria.fr/meije/verification/
[31] Barros, T.: Formal specification and verification of distributed component systems. PhD thesis, University of Nice - Sophia Antipolis (November 2005)
[32] Lang, F.: Exp.Open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
[33] Rausch, A., Reussner, R., Mirandola, R., Plášil, F.: The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008)

Chapitre 6

Specification Languages

6.1 Summary

We have explained in the introduction how we decided to base our model generation methodology on some sort of specification language, rather than static analysis of Java code. The benefits are two-folded :

- the model generation is more precise, and more properties can be proven, than relying on code analysis ; and "safe by construction" code generation methods can be envisaged,
- the verification activity can be performed much earlier in the development process, allowing one to find problems, and to build a reliable software architecture, as soon as possible.

There are many existing modeling formalisms, and specification languages, for distributed applications. Is there one fitting our needs ? We did not want to invent "yet another formalism", especially because we want our methods to be accessible to non-specialists developers. We have explained in Section 5.1 our early attemps to use UML for behavioural and architectural description of GCM components, and how we reached 2 opposite decisions for these two aspects :

1. At the level of architecture descriptions, we finally decided that the differences were too important, and it was too complicated, and not really efficient, to define a specific UML profile for describing GCM components. The next step was to define a specific graphical formalism more natural for this goal. We implemented it as the VerCors Component Editor (VCE), and built a number of case-studies with this formalism [C-04b], [32]. This graphical formalism is also the one used in the definition of the GCM ADL standard (ETSI [S-09]), but the formalism also includes GCM features that are not (yet) in the standard : complex and structured component membranes, and non-functional components. We recently defined [79] an extension of this formalism for describing usual parametrized topologies of distributed components, including arrays, rings, pipelines, or matrices.
2. At the level of behaviour descriptions, it is less difficult to adapt existing graphical formalisms to our needs. Basically, activity diagrams, or state-machine diagrams from UML would be suitable. The choice of an adequate formalism (and the corresponding tool) is planned in our short-term objectives.

However it is usually considered that pure graphical editors are not really convenient when defining real-size models, and that textual formalisms are better suited. In this line of research, we have defined in Antonio Cansado PhD thesis [32] the Java Distributed Components (JDC) specification language, that addresses in a

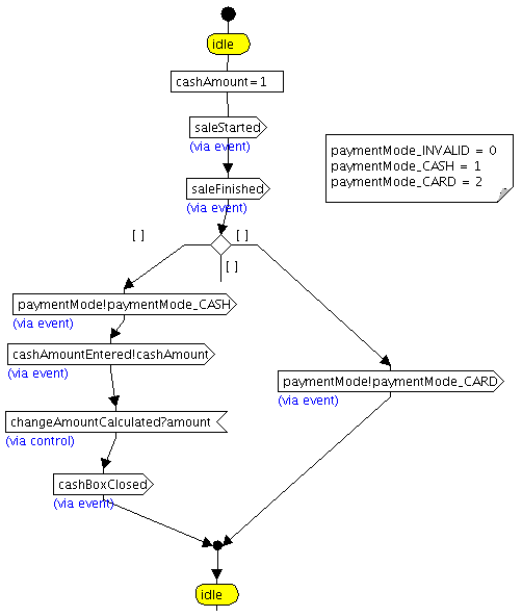


FIG. 6.1 – A State machine from the CoCoME case-study

coherent manner both the behaviour and the architecture aspects of GCM components. This language is described in the paper presented at FACS 2008 [C-08c], included here.

Last but not least, we need formalisms to express the requirements, or the properties, that will constitute the specification of our components and applications. We have been using different kinds of (action-based) temporal logic dialects, from ACTL [43] and regular μ-calculus [62, 69]. There are also means to express these logics using higher level constructs, easier to use by non-specialists, as in *specification patterns* [66], but also using some symbolic version of automata, expressing the acceptance of properties. More recently MCL [70] has been created as an extension of the regular μ-calculus, allowing formulas to include manipulation of data variables, in a manner consistent with their usage in the system definition. MCL is not yet available in the distributed version of CADP, but it provides a rigorous way to express properties with data, and we are starting to use it in our recent publications.

Discussion and Perspectives

We haven't started implementing the JDC specification language. The main reason is that we think that its architecture and abstract data part are usable, but the concurrency/behaviour part is not satisfactory in its current version. In fact there is little provision in JDC for specifying concurrency within a component in an abstract way, without giving explicitly an architecture. Moreover, the current trends towards managing multi-core processors, and optimizing applications running on such infrastructures, demand specific extensions of our behaviour specification model (see Chapter 8).

The other important research theme here is the so-called "Correct by construction" code generation method. The idea is to specify the system, prove that the specification is correct, and then generate (Java) code skeletons guaranteed to conform to the specification. We have started implementing this idea at the architecture level : the VCE editor generates ADL files implementing the component

architecture. For the other parts, we cannot expect to generate complete code, because we only have "abstract data" available in the JDC code, but more fundamentally because we do not want the behavioural specification to include all the functional logic of the application : this would not be manageable. Instead, we want to generate only the minimal code skeleton required to **guarantee** the behavioural properties of the application. Then the correctness may be based on restrictions on code modifications, together with some form of runtime condition checking (see Chapter 8).

6.2 Paper from *FACS Workshop, June 2008*

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 260 (2010) 25–45
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Unifying Architectural and Behavioural Specifications of Distributed Components

Antonio Cansado[a], Ludovic Henrio[a], Eric Madelaine[a] and Pablo Valenzuela[b]

[a] *INRIA Sophia-Antipolis, CNRS, I3S, UNSA. 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France. First.Last@sophia.inria.fr*

[b] *Universidad Diego-Portales, Ejército 441, Santiago, Chile, pablo.valenzuela@inf.udp.cl*

Abstract

We present a novel specification language called JDC to be used at design phase of distributed components. The extensive seek for asynchrony in distributed components demands new techniques for its specification that have not been addressed before. We propose to focus the specification on its data-flow; this allows to reason about inter-component synchronisations produced by a data-driven synchronisation model. The language is endowed with enough formality so it allows a constructive approach; it allows the generation of behaviour models which can be model-checked, and the generation of code skeletons with the control flow of components. Globally, this approach aims at generating components with strong guarantees w.r.t. their behaviour.

Keywords: Hierarchical components, distributed asynchronous components, formal verification, behavioural specification, model-checking, specification language.

1 Introduction

Component-based software development (CBSD) has emerged as a response from both the industry and the academy for dealing with complexity and reusability in software. The main idea is to clearly define interfaces between components so that they can be assembled and composed in several contexts.

Unfortunately, software engineers often face non-trivial runtime incompatibilities when assembling off-the-shelf components. These arise due to an inadequate (or nonexistent) dynamic specification of the component behaviour. In fact, only few state-of-the-art implementations of component models take into account dynamic compatibility. The component models SOFA [22] and Fractal [8] can be specified using "behavior protocols" [22], or (for Fractal) with our pNets formalism [3]. Other component models such as CORBA Component Model [21] only check interface type-compatibility in order to realise a binding. Types are defined in an Interface Description Language (IDL).

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2009.12.030

A major originality of our work is that we target distributed component systems communicating by asynchronous method calls with futures, concretely in the frame of the Grid Component Model (GCM) [15]. The GCM is a novel component model defined by the european Network of Excellence CoreGrid. The GCM is based on the Fractal Component Model, with extensions addressing Grid computing. From Fractal, GCM inherits a hierarchical structure with strong separation of concerns between functional and non-functional behaviours. The extensions to Fractal come from the fact that in Grid computing components are deployed over thousands of nodes, so scalability plays a major role.

Even if there are many specification languages in the literature, none fits well in the context of distributed components. In the GCM, most difficulties come when specifying the synchronisations. From a practical point of view, we focus on a reference implementation of GCM in Java: GCM/ProActive. In GCM/ProActive [11], components communicate through asynchronous method calls with futures. Futures act as placeholders for promised return values. Synchronisations happen upon data access on a future, and futures can be transmitted in remote method calls to other components; finally, almost any object in the program can be a future or not in a transparent way. Such transparent futures alleviate the programmer from synchronisation difficulties, allow for separation of concerns (the source code can be really independent from the physical infrastructure), and give optimisation opportunities at the middleware level. On the other hand, specifying and/or inferring about synchronisations becomes more complex; we need to provide help to the programmer. To our knowledge, no specification language has been proposed within this context.

Our approach in [10] was to attach the behaviour of components as part of the architecture specification, defined in terms of *Parameterized Networks of Transition Systems* (pNets) [3]; pNets is a powerful model that expresses parameterized topologies of processes communicating with value passing. Using pNets, we showed how to synthesise the behaviour of distributed components; however the formalism is too low-level to be used as a specification language, and lacks of the high-level concepts particular to the different contexts in which we want to use it.

Related work

In the same spirit, "behavior protocols" [22] is an ongoing research project that seeks formal specifications of components. They opt for simplicity rather than expressivity, for example "behavior protocols" uses a simple regular-language to describe traces of the component behaviour. This allows them to check for behavioural mismatches, however they only take into account a limited abstraction of the data-flow.

STSLib [18] provides a formal component framework that synthesises components from symbolic protocols in terms of Symbolic Transition Systems (STS). Just as pNets, STS concisely represents infinite systems, however, STS relies on Abstract Data Types (ADT) which are more expressive than our Simple Types (see Section 2.3), but less intuitive for software engineers. Both formalisms rely on (N-ary) synchronisation vectors, but in STS they are static whereas in pNets they are dynamic;

as shown in [3], this allows us to express reconfiguration in a natural way: rebinding a set of interfaces is seen as a change in the synchronisation vectors. STSLib synthesises components based on their STS protocols; a controller interprets the STS protocol and data from the ADT is implemented (and generated) in Java. The communication in STS components is rather low-level; both emitter and receiver must agree exchange a message, although there is no clear notion of required nor provided services.

Sensoria [1] is another project which provides a mathematical framework for component interaction. It targets Service Oriented Architectures (SOA) such as Web Services and SCA (Service Component Architecture [7]). Their approach is akin with "behavior protocols", specifying the allowed interaction within the system. Our approach is closer to the programming model, expressing *what* the component does to later infer *which* are the interactions.

Contribution

The originality of our work is to focus on service invocations, and implicit synchronisation by the mean of futures. We will show that the data-flow and the access to the transmitted results implicitly set the synchronisations. This approach provides a high-level and powerful abstraction for the programmer that is close to the programming model.

Instead of proving that legacy code is safe, in this paper we take a constructive approach similar to [14,18]. The idea is to specify the system, prove that the specification is correct, and then generate (Java) code skeletons guaranteed to conform to the specification. pNets is left as the underlying formalism that interfaces with model-checkers, and the programmer uses a high-level specification on top of pNets. The language is called *Java Distributed Components* (JDC for short).

Paper structure

The paper is organised as follows. Section 2 discusses the foundations of the specification language. Then, Section 3 illustrates how components can be described and composed using an architecture specification. In Section 4, we define the black-box behaviour of a component, that abstracts the internal details of a component. Section 5 specifies abstractions of user types. Finally, Section 6 explains how to generate both behavioural models and code skeletons from our specification language.

2 Foundations of the Specification Language

Distributed components tend to be coarse grain units of composition, and are often loosely-coupled. In the following we present a specification language in the form of an extension of a subset of Java for specifying these components. The language includes both the architecture and the behaviour definitions, and is endowed with enough formality and control-flow information so that we are able to:

- on one hand check the correctness of the system (Section 6.2): we build a behaviour model that can be model-checked against temporal formulas;
- on the other hand generate safe components (Section 6.3): we generate the control code of components that is guaranteed to respect the specification.

We opt for a Java-like language for several reasons; (i) it is close to the target expertise of engineers, using common syntax such as method calls and data classes; (ii) it allows to embed part of the specification within the code skeletons; (iii) it uses the same datatypes as in the implementation, guaranteeing that operations on the datatypes are directly useful without modification.

2.1 Background on Distributed Components

A recent approach to deal with distributed components on Grids is provided by ProActive [11], the reference implementation of the GCM. Components communicate through asynchronous method calls. A method call creates a request in the queue of the target component, and a future on the caller side as a placeholder for the result. These futures may be transmitted between components, no explicit instruction deals with futures, neither for creation nor for access, but access to the queue is explicit. `serve(`*method*`)` is used to select methods from the queue.

ProActive guarantees that, once the promised value of a future is known, it is transmitted to every component that has received a reference to it. Moreover, the various strategies used for transmitting the future values are proved not to change the component behaviour. A precise operational semantics of ProActive is given by the ASP-calculus [12]. These results inspire our specification language to adopt futures in order to decouple components.

Using transparent futures in the specification language brings the same advantages as in the programming language: the system designer doesn't have to wonder if a variable might contain a future; or more precisely, no explicit synchronisation mechanism is needed for variables that may sometimes contain a future. This extends reusability of specifications as they may fit several contexts, where values are remotely computed, or come from local instances. A drawback of transparency of futures is non-determinism; it is in general not statically decidable whether a variable is a future or not at a given point of the program. However, additional synchronisation can be specified, ensuring that, after synchronisation upon a variable, this variable is known to be value, or a future with a filled value.

Dynamic reconfiguration is supported in the GCM, however, it is not yet considered in JDC. In [3] we proposed models, based on our pNets, to handle reconfiguration of components. We plan to extend the language towards this direction.

2.2 Decomposing the Behaviour into Services

The functional behaviour of the component is an abstraction of the control-flow, some elements of data-flow, and access to data. Concretely, for the distributed components we deal with, the interesting events are:

- *Remote method calls*, these represent communication between components. A remote call is always an asynchronous, it creates a request in request queue of the callee component, and it creates a future in the caller for dealing with the promised result. Remote calls are identified by calls on client interfaces.
- *Future flow*, these represent the creation of implicit communication channels between the component that computes the value of the future, and the component that receives the reference to the future. The future flow can be identified by tracking future objects in parameters and results of remote method calls.
- *Data-access*, these trigger synchronisations between components. They are identified using static analysis, or given explicitly within the specification.

The first part of the component specification is called the *service policy*; it defines how a component selects requests depending on its internal state, and any behaviour the component triggers by its own. This is a rough specification of the component protocol, however, it gives the user a good idea of how the component should be used. For instance, the specification may specify that a component must serve requests in a particular order.

The second part of a service specifies what each request exposed at the service policy actually does. This behaviour is defined by a Java-like language that is very close to the programming model we want to specify. In there we include an abstraction of the control and data flow, remote method calls done within the service method, and access to data. Although it requires static analysis to infer the behaviour, it is easier than in standard Java; remote calls are easily identified by calls on the component's client interfaces; future creation points are identified as the results of these calls; there is no concurrency within the service method; and there is no exception handling (for the sake of asynchrony).

2.3 Datatypes and Abstraction

The datatypes used in JDC are standard Java classes. This way the code skeletons obtained by our generation tools will be directly usable. On the other hand, arbitrary datatypes often have large (possibly infinite) domains which can't be model-checked directly. The kind of behavioural properties we seek only require an abstraction of these datatypes. Therefore, whenever verification is desired, the specification includes as well an abstraction of the user types that allows to derive a simpler specification.

The abstraction keeps solely data influencing the control-flow and the synchronisations, however, it must preserve the behavioural properties in the sense of Cousot's abstract interpretations [16]. If abstractions are finite and constitute abstract interpretations of the initial parameter domains, then the model is finite. Following [13], we build an abstract interpretation of the system behaviour, from abstractions of the domains of the program variables; this construction can be used for finite model-checking as it preserves safety and liveness properties.

The abstractions are mappings from user types to predefined first order datatypes (*simple types* for now on). Simple types themselves are provided as Java

classes, and as a particular case, can be used in JDC programs. They are: point (or singleton), booleans, enumerated types, integers, intervals of integers, strings, records of simple types, arrays of simple types.

In our work we decompose the abstraction in two steps: the first maps concrete types to potentially infinite *simple types* allowing us to generate parameterized *pNets* models. From pNets, we can apply many different proof methods, including inductive theorem proving techniques, that can address a large family of properties. The second step is based on finite partitions of parameter domains that depend on each set of properties to prove. In this case, the abstraction produces finite pNets on which we can use explicit-state model-checkers.

Finally, our abstractions must consider futures. Even if a variable has insignificant values, access to the variable may still trigger synchronisation. This makes the choice of a good abstraction tricky, and some variables are only kept within the abstraction in order to signal eventual access on them. In other words, these variables have an abstract domain with 2 values *filled* or *non-filled*.

3 Architecture Specification

In the next sections, we present elements of the abstract and concrete syntax of JDC. Each box defines a piece of JDC syntax, using: keywords in bold (e.g. **component**); terminal symbols written between simple quotes (e.g. '{'); non-terminal symbols in monospace (e.g. `Services`); optional expressions with square-brackets (e.g. [`expr`]); choices with | (e.g. `expr1` | `expr2`); concatenations of zero (resp. one) or more expressions with * and + (e.g. `expr`*, `expr`+); and identifiers: 'id' .

3.1 Defining a Component

The definition of a component type comprises its external interfaces with both provisions and requirements, and a specification of its behaviour. The behaviour is either given by a black-box specification in the form of a set of *Services* (Section 4), or by a composition of components, also called *Architecture* (Section 3.2), or even by both.

`Component` →	**component** 'id' '{'	≪component definition≫
	external interfaces	
	`Interface`*	≪set of interfaces≫
	[`Services`]	≪black-box description≫
	[`Architecture`]	≪content description≫
	'}'	
`Interface` →	**server** | **client**	≪interface role≫
	interface `InterfaceType` 'id' ';'	≪type and name≫

Each interface in a component has a role (either server or client), a type (a Java interface as in most IDLs), and a name. The interfaces defined within the context of the component definition are *external interfaces* and can be bound to the environment. Interfaces determine both provided and required services of a

component; provided services are defined by server interfaces, and required services are defined by client interfaces.

3.2 Composing Components

The composition of components is done within the *architecture*. It exposes the content of a component by means of its subcomponents, its internal interfaces, and the bindings. The subcomponents are named and typed, the type being given by either an external component definition, or by an inline definition. The bindings connect two interfaces among the component's internal interfaces and the subcomponents' external interfaces.

`Architecture` →	**architecture**	
	contents	
	`Subcomponent`*	≪set of subcomponents≫
	internal interfaces	
	`Interface`*	≪set of interfaces≫
	bindings	
	`Binding`*	≪set of bindings≫
`Subcomponent` →	`ComponentType` 'id' ';'	≪named subcomponent≫
	| `Component`	≪inline definition≫
`ComponentType` →	'id'	≪reference to a type≫
`Binding` →	**bind** '(' `SourceItf` ',' `TargetItf` ')' ';'	≪binds a pair of interfaces≫

In the GCM, the relation between an internal interface and an external interface of a component is arbitrary: *interceptors* can transform or intercept any incoming invocation. For simplicity, in this paper, we assume that there is an exact match for each pair of external-internal interfaces (interfaces that have the same type and name, but with opposite roles); and that invocations on an external (resp. internal) server interface is directly forwarded to the corresponding internal (resp.external) client interface.

3.3 Example

The CoCoME example [9] was implemented using GCM / ProActive. It is a Point-Of-Sale system, in which the cash desk deals with the sales. The Cash Desk and its hardware controllers are implemented as components, depicted in Figs. 1 and 2.

4 Behaviour Specification

When designing a system, the designer would like to adopt a top-bottom approach: specifying first the behaviour of a component before going down into its architecture. Thus, we also propose to specify directly the behaviour acceptable by the interfaces; this is called a *black-box* behaviour of a component. Of course different *architecture* definitions can match the same component *black-box*. In this paper, we leave the equivalence (or preorder) between a component *black-box*, and its implementation (*architecture*) unspecified. Many existing work can apply, starting with all notions

```
component CashDesk {
  external interfaces
    server interface ApplicationIf appIf;
    client interface ScannerIf scannerIf;
    // ... external interfaces
  architecture
    contents
      component Application application;
      component Scanner scanner;
      // ... controllers
    internal interfaces
        server interface ApplicationIf appIf;
        // ... internal interfaces
    bindings
      bind(this.appIf, application.appIf);
      // ... bindings
}
```

Fig. 1. *Architecture* specification

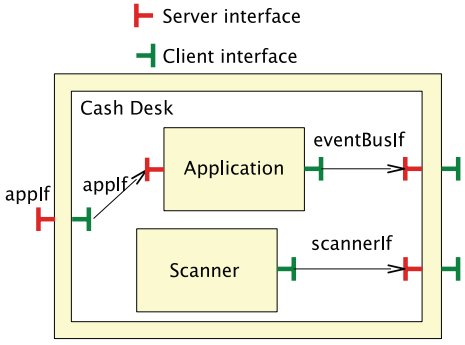


Fig. 2. Equivalent schema

of simulations and bisimulations inherited from process algebras. They have to be adapted to our component model though, e.g. in a way similar to the component substitutability relations of [23].

In GCM there are two kinds of components, *primitives* that are atomic components, and *composites* that are components composed of other components. Primitives are monothreaded, and concurrency is introduced by composites. The concurrency in JDC is specified by a set of concurrent services within the *Services* block. Each *service* denotes a sequential process with its own set of local variables. A sequential process is split into the *service policy* that defines the high-level protocol of the service, and a set of *service methods* that details the behaviour of the methods exported by the component.

```
Services →   services
               Service+                ≪one or more concurrent services≫
Service →    service  '{'
               LocalVariableDecl*      ≪variables of the component≫
               policy  '{' Policy '}'  ≪service policy≫
               ServiceMethodDecl*      ≪service methods (exported)≫
               LocalMethodDecl*        ≪local methods (not exported)≫
             '}'
```

4.1 Service Policy

The service policy defines how incoming requests are selected from the queue depending on the internal state of the component, and any behaviour triggered internally. It is given by (non-deterministic) state-machines, expressed using regular expressions. The actions can express *reactive* or *active* behaviour.

The *reactive* behaviour defines which kind of methods to select, and in which order to pick them from the queue. This represents work that depends on the requests at the component's request queue. As an example, `serveOldest(itf.m1, itf.m2)` selects from the queue the oldest request on `m1` or `m2`; if none of them is in the queue, the service blocks until one of them arrives. Then, the request is served,

`Policy` →	`ServeMode '(' [ Filter ] ')'`	≪reactive service≫
	`| MethodCall`	≪active service≫
	`| Policy ';' Policy`	≪concatenation≫
	`| Policy '|' Policy`	≪choice≫
	`| Policy '*'`	≪Kleene closure≫
`ServeMode` →	**serveOldest** | **serveYoungest**	≪request queue accessor≫
`Filter` →	`| InterfaceName`	≪any method in this interface≫
	`| InterfaceName '.' MethodName`	≪this method≫
	`| Filter ',' Filter`	≪a list of filters≫

i.e., the control is delegated to the service method representing the request.

Additionally, an *active* behaviour denotes spontaneous behaviour, i.e., some work that is done without being requested. In our example, a component in charge of the scanner sends signals to the application component whenever a product is scanned. The signals take the form of method calls on the application components. For the scanner component, this behaviour is spontaneous as the interaction with the physical scanner is abstracted away.

The service policy is the only block authorised to access the queue. Basically, this ensures that the code generated for the service policy will be complete w.r.t. how the component provides services. Moreover, the state-machines are precise enough to ensure that the code generated will be the final implementation of the `runActivity()` method of a GCM/ProActive component, and no other method will access the component's request queue. More details are discussed in Section 6.3.

An example of a *Service* definition is found in Fig. 3. We give part of the behaviour of the cash desk application. It has a single service (the component is indeed monothreaded), and it is mainly reactive; it responds to incoming events in a FIFO order.

4.2 Concurrent Behaviour

A primitive component can be specified by a single *Service*. This specification fits as well in a composite component with a pipeline of subcomponents inside. In any of these configurations, two request calls are treated sequentially. However, a single *Service* cannot express concurrency as there is no explicit thread creation in JDC. Instead, concurrency of requests is defined by multiple services within a component. Each service is an independent activity serving requests in parallel, with its own set of local variables and provided services.

A drawback of this approach is that it is not possible to define interference among the services directly. That is, we must rely on an *architecture* definition that composes independent components in order to express interference. Other alternatives would have introduced more complexity to the language; moreover, the generation of the control code would have been difficult as the programming model doesn't have explicit concurrency.

```
services
   service {
      // variables of simple types
      Bool expressMode;
      public enum CashState{
         IDLE, STARTED, PAYING
      }
      CashState cashState;
      // ... other variables of
           simple types and user
           types

      // initialises the system with
            some RPC and then treats
            calls in FIFO order
      policy {
         init(); // local method
         serveOldest(applicationIf)*
      }
     // ... the service methods
  }
```

Fig. 3. Service definition of the Cash Desk Application

```
void applicationIf.barcodeScanned(Barcode barcode) {
   switch (cashState) {
      case IDLE:
      case PAYING:
         break;  // ignore signal
      case STARTED:
         Product product = cashDeskIf.getProduct(barcode
              );
         if (product == null) {
            eventBusIf.productBarcodeNotValid();
            break;
         }
         if (expressMode && products.isFull())
            __ERROR("ExceededNumberOfProducts");
         else {
            products.add(product);
            runningTotal.add(product.getPurchasePrice());
            eventBusIf.runningTotalChanged(runningTotal,
                 product);
} }    }
```

Fig. 4. A Service Method of the Cash Desk Application

4.3 Service Methods

A service method is an abstraction of a service exported by a component. It is defined by means of a subset of Java statements in which there is no exception handling, and no concurrency. This includes the relevant dataflow between input parameters and results of the method, as well as communication with required services. The service method has access to the component's variables, however, it doesn't access the component's request queue.

Java is extended to deal with component interfaces. The name of the service method is prefixed by the server interface in which it is defined. Client interfaces are accessed as usual objects but they *cannot be assigned to other variables*. This last requirement is very important to ensure that all the interactions between components are realised through the client interfaces.

An example of a service method is depicted in Fig. 4. The behaviour focuses on a cash desk that may provide an express mode for dealing with sales with a limited amount of products. When the barcode of a product is scanned, the component reacts accordingly to its internal state. Its usual behaviour is to get the product information by invoking a remote method call (`getProduct(barcode)`), add the product to a list of `products`, and update some information regarding the current sale (`runningTotal`). The specification is quite close to Java, notably the operations on the product are the ones that would be expected in a real implementation.

5 Specifying Abstractions

This section shows how to define and use abstractions of user types in JDC. One particularity is that a class may have more than one abstraction defined, each one focusing on the significant behaviour of a variable.

The abstractions ensure that we are able to generate behavioural models based on pNets. pNets allows us to interface with several verification tools; for the moment

we focus on finite-state model-checkers, but using pNets we can potentially interface with infinite-state model-checkers and theorem provers as well.

5.1 Formalisation of an Abstraction

A class is a tuple $\mathcal{C} =< \overrightarrow{m}, \overrightarrow{f} >$, where $\overrightarrow{m} = \{m^i(\overrightarrow{a}) : \tau^i\}$ are the methods of $\mathcal{C}$; $\overrightarrow{a} = \{a^j : \tau^j\}$ are the method arguments; and $\overrightarrow{f} = \{f^k : \tau^k\}$ the fields.

An abstraction of $\mathcal{C}$ is a class $\mathcal{C}_\mathcal{A} =< \overrightarrow{m_\mathcal{A}}, \overrightarrow{f_\mathcal{A}} >$, where each public method $m(\{a^j : \tau^j\} : \tau)$ of $\mathcal{C}$ has one or more abstract method $m_\mathcal{A}(\overrightarrow{a_\mathcal{A}}) : \{\tau_\mathcal{A}\}$ with $\overrightarrow{a_\mathcal{A}}$ the abstract arguments, which domains are sets of values in the abstractions of classes τ^i, and the result is an abstract value in the abstraction of class τ.

For defining what is a good abstraction of the domains of the variables in the specification, we need to identify:

- where in the specification are the "variables of interest" – those used in the properties to be proved;
- what are the significant values of these "variables of interest" – these will determine their abstract domain;
- which other variables in the program influence (through control-flow and data-flow) the "variables of interest" – these other variables will also have a non-empty abstract domain.

For each of these significant variables, we must attach an abstract type in the following manner:

- for each public method m of $\mathcal{C}$, abstract versions $m_\mathcal{A}$ are provided that capture the accesses on the class variables, accesses on the variables passed as arguments, and relevant results of them.
- the fields of the concrete class that are of interest are included as a record. The domains of these fields are such that they are precise enough to hold the property to prove. This is done recursively in order to find the abstractions of the other variables of interest.

5.2 Using Abstractions

An abstraction in JDC is similar to a Java class, with extensions to deal with non-determinism and data abstraction. An important notion is that we may have to use different abstractions for different variables of the same concrete type, within a given program. This means that in the abstract program, we may need different versions of the abstract operators, depending on the abstract types of the arguments. For example, if the concrete program has variables `x:Int, y:Int` then the abstract program may have `x:Sign, y:[0..3]`, and we may need to define the `+` operator for arguments in `Sign and [0..3]`. We solve this problem in two phases: we define a library of abstract classes (here `Sign` and `interval` as abstractions of `Int`, with the standard abstract operators in each (e.g. `+ : Sign*Sign -> Sign`); these libraries can be defined in a generic way, and reused easily. Then for a specific program,

we define abstract classes that inherit from the required library abstract classes, and define additional abstract operators depending on the specific abstraction of variables, and of the occurrences of the operators found in the code (e.g. `+ : Sign*[0..3] -> Sign`).

`Abstraction` →	**abstraction** 'id' **of** 'id' '{'	≪datatype abstraction≫
	`Field`*	≪local variables≫
	`Constructor`*	≪abstract constructors≫
	`Operator`*	≪abstract operators≫
	'}'	
`Field` →	`Type` 'id'	≪type and name of variable≫
	[**abstracted as** `Type`]	≪local mapping of a type≫
`Operator` →	`Type` 'id' '(' `args` ')'	≪signature of concrete operator≫
	[**abstracted as** `Type` 'id' '(' `args` ')']	≪signature of abstract version≫

The fields within an abstraction are variables of type *simple type*, or any other usertype provided with an abstraction. The latter can be given by a unique global abstraction for the type, or by an inline abstraction that selectively determines the abstraction for the type.

The operators are abstract versions of the class methods, that capture the behaviour of interest for a variable. It is possible to have multiple versions of the same operator, each one taking different abstract versions of the arguments and return types. Similarly, the same applies to constructors.

It is often useful (or required) to underspecify what are the results of an expression, possibly as the result is a set of abstract values. The language includes for that two non-deterministic operators; the first, called `ANY`, non-deterministically returns any element of the abstract domain; the second, called `ANYELEMENT`, non-deterministically selects an element from a list.

Moreover, it is often not possible to statically know if a variable refers to a value or to a future. The safe assumption is to consider such variable as *possibly future*. In here, we exploit that a *non-future* variable is semantically equivalent to a *future* variable with filled value. Nevertheless, the user must keep in mind that some traces in the specification may never occur in a concrete implementation. A solution can be then to make the specification more precise by enforcing more synchronisation on a variable (by means of `touch()`). After the synchronisation, the variable is known to be *non-future*. `touch()` synchronises on the variable without describing which operations are applied. This allows details of the implementation to be filled-in later without changing the synchronisations occurring in the system.

5.3 Example

```
abstraction ListProducts_A of ListProducts {
   enum ListState { EMPTY, OK, FULL }
   List<Product> products abstracted as
        ListState;
   ListProducts() abstracted as
        ListProducts_A() {
      products = EMPTY;
   }
   Bool isFull() { return (products==FULL); }
   Product get() abstracted as Product_A get
        () {
      switch(products) {
         case EMPTY:
            return null;
         case OK:
            if (Bool.ANY())
               products = EMPTY;
            return Product_A.ANY();
         case FULL:
            products = OK;
            return Product_A.ANY();
   } }
   void add(Product product) abstracted as
        void add(Product_A product){
      product.touch();
      switch(products) {
         case EMPTY:
            products = OK;
            break;
         case OK:
            if (Bool.ANY())
               products = FULL;
            break;
         case FULL:
            break;
} } }
```

The example above illustrates the use of a data abstraction influencing the control-flow. A short-sale must not exceed a maximum number of products, but there is no constraint on the type of products. Therefore, the abstraction of the product list must be precise enough to take into account whether the maximum has been exceeded or not, and can abstract away the product information.

The abstraction for the product list has no counter. Instead, it focuses on the states the list can have: the list is either `EMPTY`, `OK` or `FULL`. This abstraction is imprecise w.r.t. the number of products it has, so actions on the list are non-deterministic. Adding a product from an `EMPTY` state never reaches the limit for a short-sale, however, from an `OK` state it may (the state change to `FULL` is non-deterministic). Note that the context guarantees that we never call `add()` when the list is `FULL`.

The abstraction for the `product` is such that we are able to signal access upon the variable. This is necessary as the `product` may be a future; indeed, in Fig. 4 `product` is the return of a remote method call and thus can be a future. Therefore, the `product` is abstracted as a Singleton domain (`Product_A`) such that the access is signalled by `touch`.

6 Work in Progress

The middle term aim of this work is to create code with a guaranteed behaviour. It is therefore natural to start by checking the behaviour of a component, and then to generate code skeletons for the components.

6.1 Finding Abstractions

Defining abstractions can be burden without a tool support. For developping this kind of tool a first step is to characterise what is a good abstraction. It surely depends on the property to prove, but there are a couple of general ideas that support some automatising of the abstractions.

Using static analysis, the variables used in the property will signal which are the

"variables of interest". The abstract domain for these variables is such that if there is a non-deterministic choice affecting the property, then the abstraction must be refined. There are tools like Bandera [17] that take this approach. Bandera defines a family of abstractions for a variable and lets the system find the least precise one that still holds the property. This work must be extended, though, to take into account futures. At least one needs to find the set of variables that may contain a future in any of their subfields. This leads us to the set of variables that must have a non-empty abstract domain as well. Moreover, this gives us the most abstract structure a variable can have for its type, i.e. a record with a field (or recursively subfield) for each of these variables with non-empty abstract domain.

6.2 Behaviour Model Generation

Building the behavioural model requires to abstract the JDC specification into a corresponding specification with only simple types. This is done by replacing each variable of user type by its abstraction. Then the pNets model will create:

(i) for each service, a storage for each of its local variables. A storage is a parameterized Labelled Transition Systems (pLTSs) that stores the variable state, and that exports actions *set* and *get* for accessing the variable. These storage are synchronised with all the pLTSs of the service methods and the service policy.

(ii) pLTSs for specific library elements of JDC, e.g. request queues, and proxies for futures. The latter requires dataflow analysis of the futures flow – in [4] we have defined a similar procedure.

(iii) a pLTS for each service policy. The service policy is a state machine so the transformation is straightforward. The *reactive* behaviour is transformed into two actions, one synchronised with the queue, and another that fires the affected service method. Similarly, each *active* behaviour is transformed into an action that fires the method directly.

(iv) a pLTS for each service method. This requires static analysis of the pseudo Java code of the abstract specification.

(v) synchronisation structures (pNets) for relating these pLTS. Each component is modelled by a pNet that synchronises the actions of the pLTSs – the model was previously shown in [3].

(vi) a tree of pNets modelling the architecture of the components. Each branch is the pNets model of a component, where its branches are the pNets of its subcomponents – the model was previously shown in [3].

6.3 Code Generation

From the JDC specification, it is possible to generate GCM/ProActive code skeletons with the control code of the components. Java code is only generated for sequential components, so concurrent components must be provided with an architecture that decomposes the behaviour into sequential components. The ProActive

middleware is adequate because it supports distributed components that communicate with first-order futures. We base our method on the following steps:

For each *Architecture* specification of a component, the compiler generates a composite component. The composite architecture is expressed with the GCM ADL (Architecture Description Language). The composite ADL defines the component type and its content based on the ADLs of other subcomponents, bindings and the IDLs of the interfaces.

Each *Service* denotes a sequential component, and therefore its natural implementation is a primitive component. An ADL is also created for defining the component type, as well as a reference to its Java implementation. A code skeleton is generated for the latter with the control flow. The code is a translation of the JDC's black-box specification based on:

- each service method in JDC is a public method of the component. We rely on the strong functional behaviour encapsulation of GCM for this matter, and that every possible method call and data-usage appears in the black-box specification.
- all data types are created, but these will need to be modified by the programmer to give implementation details.
- the service policy is implemented as a state machine within the ProActive's `runActivity()` method. This method dictates the initial activity of the component and we use it to orchestrate the access to the queue and to serve requests.

7 Conclusion

We aim at safe-by-construction components. Our approach is to define the architecture, the behaviour, and an abstraction of data within the specification language. The specification is formal enough in order to generate behavioural models that can be model-checked, and to generate code skeletons that include the control code of components.

More specifically, our contribution in this work is:

- A high level specification language for distributed software components, called JDC, that includes architectural, behavioural, and data parts. The behaviour of a component is given as a set of services; the details of a service are given in a Java-like language that makes easy to specify the control and data flow.

  The data part is an abstraction of the final application data classes. It must be designed by the developer as a compromise between verification and implementation concerns: precise enough to keep track of domains of variables affecting the control and data flow, but abstract enough to allow model-checking.
- Procedures for producing a hierarchical behaviour model, in pNets format, on one side, and code skeletons, in GCM/ADL and Java, on the other side.

This work builds on the GCM, however, at the moment only a small subset of it is addressed. We plan to extend the language to cope with other interesting features, such as group communications and non-functional aspects (dynamic re-

configuration). Currently we have no tool support for the JDC language, except for a graphical version in the form of an Eclipse editor of the architecture part. Nevertheless, we plan to have a first prototype for the full language by the time of the workshop.

References

[1] Sensoria webpage. http://www.sensoria-ist.eu.

[2] S. Ahumada, L. Apvrille, T. Barros, A. Cansado, E. Madelaine, and E. Salageanu. Specifying Fractal and GCM Components With UML. In *proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC'07)*, Iquique, Chile, Nov. 2007. IEEE.

[3] T. Barros, R. Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, accepted for publication, 2008. also Research Report INRIA RR-6491.

[4] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed Java objects. In *Forte'04 conference*, volume LNCS 3235, Madrid, Sept. 2004. Spinger Verlag.

[5] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model checking distributed components : The Vercors platform. In *3rd workshop on Formal Aspects of Component Systems*, Prague, Czech Republic, Sep 2006. ENTCS.

[6] F. Baude, D. Caromel, L. Henrio, and P. Naoumenko. A flexible model and implementation of component controllers. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, pages 12–23, june 2007. CoreGRID TR-0080 technical report.

[7] BEA Systems, IBM, IONA, Oracle, SAP AG, Siebel Systems, and Sybase. Service component architecture. Whitepaper, November 2005.

[8] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani. An open component model and its support in java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, may 2004.

[9] A. Cansado, D. Caromel, L. Henrio, E. Madelaine, M. Rivera, and E. Salageanu. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, chapter A Specification Language for Distributed Components implemented in GCM/ProActive. Springer, 2008. http://agrausch.informatik.uni-kl.de/CoCoME.

[10] A. Cansado, L. Henrio, and E. Madelaine. Towards real case component model-checking. In *5th Fractal Workshop*, Nantes, France, July 2006.

[11] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[12] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

[13] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *Int. Conference on Concurrency Theory (CONCUR)*, volume 836 of *LNCS*, pages 417–432. Springer, 1994.

[14] A. Coglio and C. Green. A constructive approach to correctness, exemplified by a generator for certified Java Card appplets. In *Proc. IFIP Working Conference on Verified Software: Tools, Techniques, and Experiments*, October 2005.

[15] CoreGRID, Programming Model Institute. Basic features of the grid component model (assessed). Technical report, 2006. Deliverable D.PM.04, http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf.

[16] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 138–156. Springer-Verlag, 2001.

[17] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[18] F. Fernandes and J.-C. Royer. The STSLIB project: Towards a formal component model based on STS. In *Proceedings of the Fourth International Workshop on Formal Aspects of Component Software (FACS'07)*, Sophia Antipolis, France, September 2007. ENTCS.

[19] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, Aug. 2002.

[20] Object Management Group. *UML 2.0 Object Constraint Language (OCL) Specification*, formal/03-10-14 edition, 2003. version 2.0.

[21] OMG. Corba components, version 3. Document formal/02-06-65, June 2002.

[22] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

[23] I. Černá, P. Vařeková, and B. Zimmerova. Component substitutability via equivalencies of component-interaction automata. In *Proceedings of the Workshop on Formal Aspects of Component Software (FACS'06)*, Prague, Czech Republic, September 2006. ENTCS.

42 A. Cansado et al. / Electronic Notes in Theoretical Computer Science 260 (2010) 25–45

A VCE: A Graphical Tool for Architectural Definitions of GCM Components

This annex presents a graphical tool called *Vercors Component Editor* (VCE) for the design of GCM-like components. Here, we present only the graphical language for defining distributed components.

Since in JDC the architectural part is basically that found in classical ADLs, VCE can be seen as a front-end for the architectural part of the JDC specification language. There are other architectural features in the GCM that were not addressed in JDC, and we provide support for them in VCE. For example, we will show how non-functional aspects can be defined, as well as one-to-many and many-to-one communications.

This graphical tool is an extension of the work presented in [2], in which we proposed a graphical syntax for describing the architecture and the behaviour of Fractal and GCM component systems, based on UML 2.1 diagrams for component structures and state machines. In our new version, we abandoned the strict definition of the UML diagrams, that were not sufficient for our needs. Our new graphical constructs can be considered as a specific UML profile for the GCM components, or as an independent Domain Specific Language (DSL) for the GCM.

The interested reader can refer to [9] for a large case-study using our previous approach; the case-study is modelled with JDC as well.

Contribution

We provide an editor with custom diagrams for the GCM. We stress on the architectural specification of both functional and non-functional behaviours. We define a meta-model for dealing with these aspects, and provide verification features for validating the design. The tool is fully integrated into Eclipse, and can interface with the GCM runtime files, namely the GCM ADL and the interface signatures.

Context

This tool models GCM components; besides usual composition, the main particularities found in the GCM are group communication (called collective interfaces), and structuring of the control part of components.

Collective interfaces provide some synchronisation and distribution capacities. There are of two kinds in the GCM: multicast, that distribute one message with its parameters to a set of destinations; and gathercast, that synchronise and gather a set of messages with their parameters. A client interface may be a multicast interface, meaning that a call toward this interface can be distributed to many server interfaces depending on the distribution used. Similarly, a server interface may be a gathercast interface, meaning that multiple client calls will be synchronised and a single call will be performed towards the service component.

The control part of a GCM is deal with by the component's *membrane*; this one is in charge of all non-functional (NF) concerns. The membrane is composed of *controllers* that implement the NF concerns. Instead of an arbitrary implementation of the membrane (as in Fractal), we structure the membrane with a composition of *NF components*; such a structure is presented in [6].

Annex structure

In Section A.1 we present the editor; first, the functional specification, then the NF specification, and finally the validation and ADL generation features. In Section A.2 we present the tool architecture, and its place in the platform. Finally, the annex concludes in Section A.3 with a summary and some future work.

A.1 The Editor

In this section we present our component editor. VCE is built as an Eclipse plug-in using code generated using the TOPCASED environment. It is uses a Model-Driven-Architecture pattern, with an Ecore meta-model at its kernel. We do not detail the meta-model in here, but basically it is built on the symmetry between structure of the content and the structure of the membrane. Moreover, it is compatible with the architectural definition of JDC, and we expect to provide the user with similar tools for the behavioural part as well.

A.1.1 Components and their Content

The editor is based on the concepts from the Fractal and GCM model, but we have significantly changed some of the graphical notations. One important change is the representation of components. Fractal sets the role of an interface based on the orientation (left/right, top/down). While these conventions make interpretation of small diagrams easier, the diagrams do not scale well. In here we rather use a more classical notation with no orientation constraints, and interface types distinguished by icons and/or colors.

Interfaces are depicted using the icons from UML component diagrams; server interfaces are shown as filled circles (e.g. interfaces I, IA, IB, IR1 in Fig. A.2), and client interfaces as semi-circles (e.g. interfaces IC, IR, IR2 in Fig. A.2). We distinguish between *external* and *internal* interfaces. External interfaces are accessible by the environment; and internal interfaces accessible by the component's subcomponents.

Multicast and *gathercast* interfaces have custom icons. These were not considered in UML, and hence it was not possible to reuse existing ones; in Fig. A.2, we show the icons `Multi` and `Gather` that represent these interfaces respectively. The interface `Multi` broadcasts incoming messages to components `A` and `B`, and the interface `Gather` gathers and synchronises calls coming from interfaces `IC` towards the component `C`.

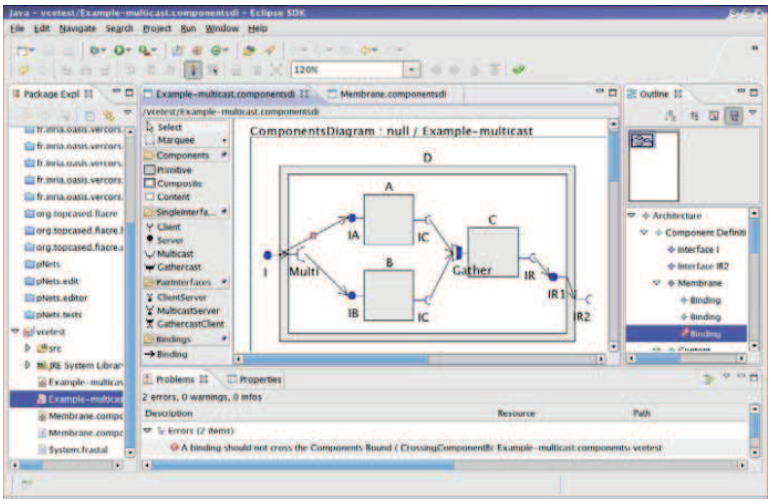


Fig. A.1. Screenshot of VCE

The *content* of a component is represented as a white rectangle inside the component, and the *membrane* is the grey area that surrounds the content. A *binding* between a pair of interfaces is presented as an arrow from a client interface to a server interface.

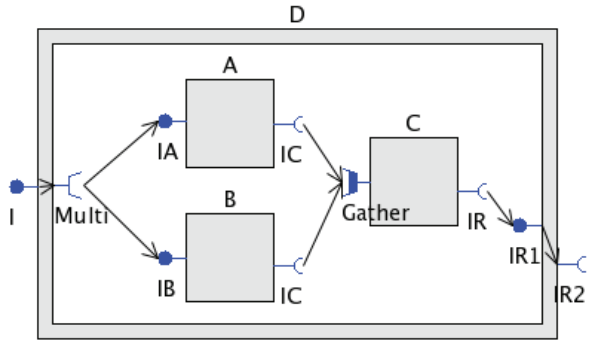


Fig. A.2. Example of composite component exposing its content

A.1.2 Membranes and Non-Functional Components

By exposing the component's membrane it is possible to control non-functional (NF) aspects. We depict the membrane with a grey area; in composites it surrounds the content, and in primitives it fills the whole figure.

The access rights of each interface is defined by marking each interface either as functional or NF. Examples of NF interfaces are `I_NF_Control` and `I_NF` in Fig. A.3. These interfaces are connected to NF components that handle the component's life-cycle.

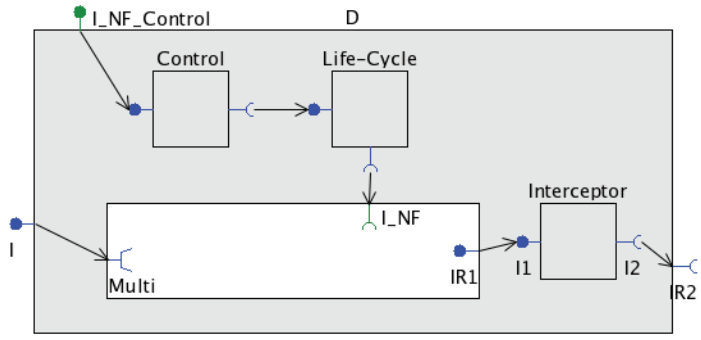


Fig. A.3. Example of a component component exposing its membrane

VCE allows the designer to intercept functional calls entering or leaving a component. The interception takes place as NF components that are connected to the component's external functional interface, and to

component's internal functional interface; they are also called *interceptors*. This is a convenient way of implementing security aspects, or to check and adapt the component protocol.

In Fig. A.2, the binding ⟨IR1, IR2⟩ forwards the calls from the internal interface to the external interface. In Fig. A.3, however, the calls are intercepted and sent to a NF component called `Interceptor`.

A.1.3 Model Validation

To ensure the integrity of the user model, we define a minimum set of invariants that every model must hold. These invariants are defined with OCL (Object Constraint Language) [20], and complement the meta-model by expressing constraints that were left undefined. By defining the rules using OCL, we let our meta-model open. This allows us, in theory, to define different set of rules depending on particular implementations of the GCM.

However, checking for interface compatibility is not feasible using OCL. This would require us to define, within the meta-models, the full interface compatibility of Java. Instead, interface compatibility will be checked independently by our tool using hand-made Java code.

The errors are mapped back to the user diagrams. Back in Fig. A.1, the designer connected the external interface with a subcomponent's interface. This error was detected, and reported as a red cross on the object that violated the constraint in the *Problems* tab, in the *Outline* view, and in the *diagram*.

A.1.4 Interfacing with ADLs

We are able to generate ADLs from these diagrams useful within GCM. In its current state, we only generate GCM ADL files with the definition of the content. The membrane is still not taken into account because the ADL for dealing with non-functional aspects is still being defined.

Moreover, we also allow the designer to import ADLs in XML. The layout, however, is manual meaning that the user needs to manually place the components in the diagram.

A.2 Tool Architecture

We are working on a verification platform called Vercors [5], Fig. A.4. We want to build a GUI that integrates the several tools found in our platform. In this context, VCE fits in as front-end to the user, and is the one responsible of interfacing with the rest of the tools. It allows the user to rapidly design the system, and provides direct tools for verifying the architectural consistency (for the moment).

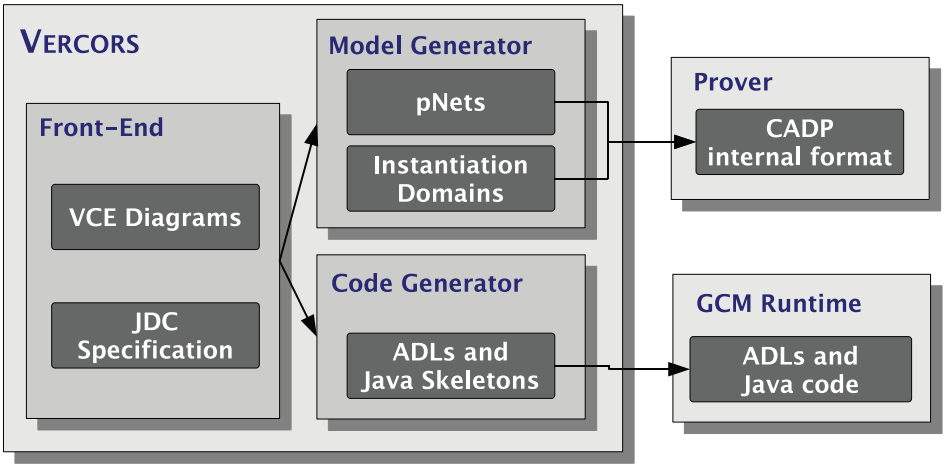


Fig. A.4. The Vercors architecture

The internal model is based on our pNets [3] formalism. Using pNets, we create behavioural models that are suitable as entry for verification tools. Our Vercors platform is using the CADP toolset [19] for state-space generation, hierarchical minimisation, (on-the-fly) model-checking, and equivalence checking (strong/weak bisimulation). We are able to verify temporal properties, including those that take into account dynamic reconfiguration of the system. The verification is based on model-checking, by translating the pNets model into a suitable input language. This is not yet supported by VCE, but we have prototypes using this approach. These will certainly be included in VCE to allow using the same GUI for the full verification chain.

We are also working on generation of skeleton code. Based on the behavioural specification of the system, we will create the control code of GCM components.

A.3 Conclusion and Work in Progress

In this annex we introduced our new graphical tool. It allows one to specify the architectural definition of GCM components, including the functional concerns, as well as the non-functional concerns. The latter takes the form of non-functional components within the component's control part – the membrane. We also support the group communication found in the GCM.

We are also working on adding behavioural specifications to VCE. These will be in the form of state-machines, and will certainly be a subset of JDC's behavioural specification. Moreover, the new tool will be fully integrated to our pNets [3] formalism. This allows us to provide a GUI for the full verification chain within our Vercors platform.

Chapitre 7

Case-studies

7.1 Summary

It should be evident that this kind of research, addressing by essence "industrial" programming languages, must be backed up by realistic case-studies. Still it was not very easy to find such significant examples in our domain : by contrast with the embedded systems or critical systems areas, where motivations for formal methods are well-established, it is not common to find large-case studies in the area of distributed systems, grid applications, or even for large-case, highly dynamic, distributed services. Among published case-studies in this area, let us mention the Airport Wifi System, that was formalized in Fractal and proved using Sofa [74], or the Scalagent example that was verified with CADP [86].

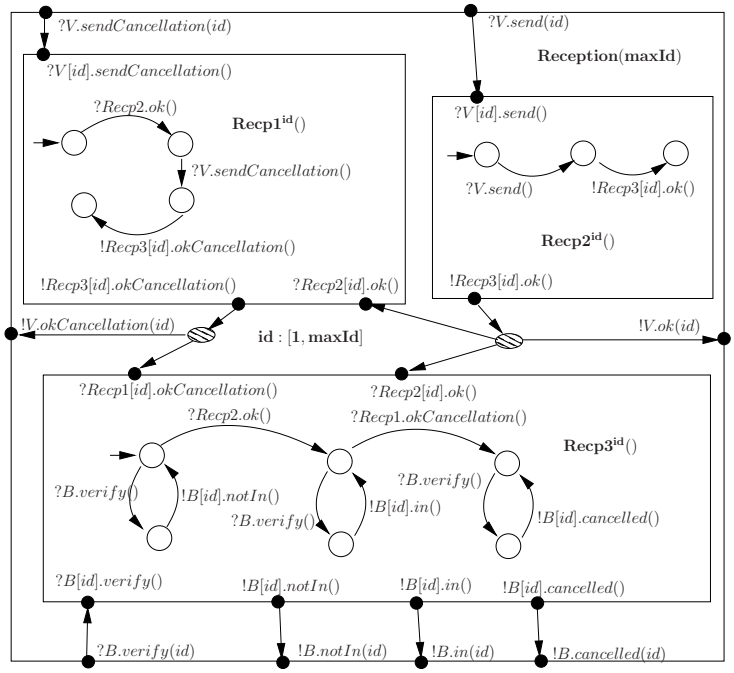


FIG. 7.1 – A pNets model from the Electronic Chilean Taxes case-study

The first realistic case-study we ran, with Tomás Barros, was the electronic tax management system that the Chilean government was studying in the years 2000-2002 [50]. This was quite a large problem, that we modeled directly at the semantic level, while developing the pNets theory (see Figure 7.1). This encoding did *not* include the modeling of active object request queues and futures, but was

already quite a large model, featuring 17 automata in 4 levels of hierarchy, and 7 parameters whose abstract domains had 2 or 3 values. The global state space, if computed by brute force would be in the order of 10^{12} states, but we were using a compositional approach, minimizing sub-system models by branching bisimulation (using the FcTools engines [25]). Using the Evaluator model-checker, we proved 7 temporal properties on this system, either safety or correctness (inevitability of responses), expressed as ACTL [43] or regular *mu*-calculus [69] formulas.

This experiment was presented at the SCCS'04 conference [C-04b], a full version with all proofs in [R-04], and also in Tomás PhD thesis [14].

Our next large case study was done in the context of a European initiative led by a number of university groups working on modeling of component-based software systems. This was the "Common Component Modeling Example" (CoCoME), consisting in a specification of a store cash-desk management system, with 15 teams applying their various modeling and analysis methodologies to the same specification. Our contribution (with Antonio Cansado and Ludovic Henrio) was naturally based on GCM, and was demonstrating the encoding in pNets of broadcast communication, and of a simple reconfiguration mechanism. It was also our first specification using both the CTTool graphical editors, and JDC code. This study was using a model based on synchronous remote method calls. CTTool automatically produced Lotos code, which was fed directly to the CADP model-checker. The CoCoME specification was encoded as a set of properties, expressed as extended Buchi automata, with acceptance or rejection states, and predicates over the abstract data types of the specification (example in figure 7.2).

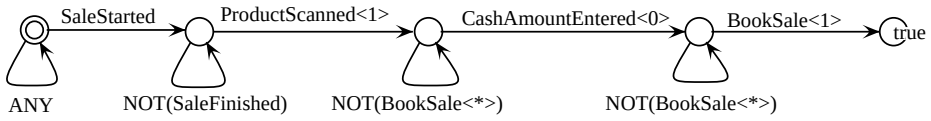


FIG. 7.2 – A formula from the CoCoME case-study

This was published as a chapter in the CoCoME book [J-08], and also in Antonio's PhD thesis [32].

Our most recent study is the subject of the two papers included in this chapter. It illustrate the modeling and verification of a group communication mechanism, on an exemple of an consensus protocol for establishing meeting agendas. This case-study is 2 orders of magnitude more complex than the previous ones, because it includes : 1) the modeling of the request queues of active objects, 2) the treatment of group communication between an object and a (fixed) number of group members receiving group requests, 3) the modeling of future proxies.

The WCSI'10 paper [C-10] defines the encoding in pNets of the behavioural semantics of queues and group communication, and shows the full structure of the example pNets model. The model was first encoded (manually) using the Fiacre format in a monolithic manner (one single Fiacre program, encoding the basic LTSs as Fiacre processes, and each level of the successive pNets synchronization as a Fiacre component). The model generation for this system was an opportunity to check the limits of the CADP distributed state generation engine (Distributor) on a cluster comprising 120 cores and 480 Giga bytes of RAM. It was also an opportunity to estimate various strategies of state generation using partial-order on-the-fly reduction methods.

The second step was to use a compositional decomposition of the system, taking care of building the state-spaces of subsystems in a properly restricted environment,

then minimizing them using branching bisimulation (see the detailed figures in [C-10, page 12]).

The last step consisted in using both fine-grain parallelism for the state-generation of sub-systems with the Distributor engine, and coarse-grain parallelism for executing independent tasks on our Cloud infrastructure. This provided us with very interesting preliminary insight on the kind of formalism that would be necessary to enable submission (by non-specialists) of complex verification tasks on such infrastructures. This was reported in [R-10], which is included here.

Perspectives and Challenges

The case-studies included in this chapter have shown that we are able to address middle-size applications including many of the basic features of and distributed components. One important exception is dynamic reconfiguration, which presents important challenges for finite-state modeling and model-checking, in particular in terms of model complexity, but also in terms of representation in the model of potential dynamic evolution (implying changes in the synchronization of events), and of the architecture management mechanics. Moreover, such "elastic" application architectures are often involving large data domains, and the definition of valid data abstraction for such systems is complex. We shall discuss ideas to attack these challenges in the next chapter.

7.2 Paper from *WCSI Workshop, June 2010*

7.3 Extended Abstract from *SAFA Workshop, Sept. 2010*

Behavioural Models for Group Communications

Rabéa Ameur-Boulifa
System-on-Chip Laboratory (LabSoC), Telecom Paristech, Sophia Antipolis, France
Rabea.Ameur-Boulifa@telecom-paristech.fr

Ludovic Henrio and Eric Madelaine
INRIA, CNRS, I3S, University of Nice Sophia-Antipolis, Sophia Antipolis, France
Ludovic.Henrio@sophia.inria.fr and Eric.Madelaine@sophia.inria.fr

Group communication is becoming a more and more popular infrastructure for efficient distributed applications. It consists in representing locally a group of remote objects as a single object accessed in a single step; communications are then broadcasted to all members. This paper provides models for automatic verification of group-based applications, typically for detecting deadlocks or checking message ordering. We show how to encode group communication, together with different forms of synchronisation for group results. The proposed models are parametric such that, for example, different group sizes or group members could be experimented with the minimum modification of the original model.

1 Introduction

Group communication is a communication pattern allowing a single process to perform a communication to many clients in a single instruction, this operation can be synchronized or optimized accordingly. Nowadays group communication is widely used in distributed computing particularly in grid technologies [27]. Objects can register to a group and receive communications handled in a collective way. Group membership is transparent to the receiver that simply handles requests it receives. Group communications are also easy to handle on the sender side because a simple invocation can trigger several communications. Communication parameters are sent according to a distribution policy; they can be for example broadcasted or split between the members of the group. Several middleware platforms and toolkits for building distributed applications implement one-to-many communication mechanisms [1, 6, 23].

This paper addresses the crucial point of reliability of distributed applications using group communications. The most frequent reliability issue for distributed application is to be able to detect deadlocks, in the case of group, a dead lock can occur for example when a member of the group does not answer to its requests while the request sender is waiting for all the results. Such an absence of response might be due to an issue in message ordering for example. In order to enhance reliability of group applications we develop methods for the analysis and verification of behavioural properties of such applications, our method can be applied with automatic tools.

A first contribution of this paper is to provide a model allowing the verification of the behaviour of group-based applications, in other words, we provide a verifiable model for group communication. We also illustrate our approach by specifying an application example, instantiating the verifiable model, and proving a few properties.

To precisely define the semantics of group communications, we focus on a specific middleware called *ProActive* [2]. *ProActive* provides a high-level programming API for building distributed applications, ranging from Grid computing to mobile applications. *ProActive* offers advanced communication

Cámara, Canal, and Salaün (Eds.)
Component and Service Interoperability (WCSI10)
EPTCS 37, 2010, pp. 42–56, doi:10.4204/EPTCS.37.4

strategies, including group communication [31, 6]. In *ProActive*, remote communication relies on asynchronous requests with futures: upon a call on a remote entity, a request is created at the receiver side, and a future is created on the sender side that will be filled when the remote entity provides an answer. What make the handling of groups particular in *ProActive* is the necessity to also gather and manage replies for requests sent to the group. Synchronisation on futures is generally transparent: an access to a future blocks until the result is computed and returned. However, synchronisation on group of futures, that represent the result of a group invocation, features more specific and complex synchronization primitives. Consequently, our model also encodes different synchronisation policies.

In [8] we have defined a parameterized and hierarchical model for synchronised networks of labelled transition systems. We have shown how this model can be used as an intermediate format to represent the behaviour of distributed applications, and to check their temporal properties. In this paper, we present a method for building parameterized models capturing the behavioural semantics of group communication systems; models are the networks of labelled transition systems, whose labels represent method invocations. The language we chose is pNets; it is an intermediate language: the models we present here should be generated, either from source code or from a higher-level specification. PNets themselves are then used to generate a model in a lower-level language that will be used for verification of the program properties. In this paper the advantage of choosing such an intermediate language are the following: compared to a higher-level language, pNets are precise enough to define a behavioural semantics, and compared to lower level languages, they provide parameterized processes and synchronization which allow the expression of the models in a generic manner.

Our approach aims at combining compositional description with automatic model generation. The formal specification consists in a labelled transition system and synchronisation networks, in which both events (messages) and processes (group members) can be parameterized and built from a graphical language. On one hand, having a well-defined semantics made the specification sound; on the other hand, having a framework based on process algebras and bisimulation semantics made possible to benefit from compositionality for specification and verification [10]. Parametric synchronisation vectors also allow us to envision the modelling of dynamic groups with members joining or leaving the group.

Related Work Some work has been done to formally verify properties in group-based applications. Some of these verifications deal with safety properties, while others remain limited to a case study. In [22] the formal verification of cryptographic protocols is proposed. It used model-checking tool to verify confidentiality and confidentiality properties. Model-checking was also used to verify behavioural and dependability properties [28]. The authors adopted Markov chains to specify the studied protocols. By using a combination of inductive poofs and probabilistic model checking [24] verified a randomized protocols. In the same way, [25] used a combination PVS theorem prover and model-checker based on timed-automata for formal verification of an intrusion-tolerant protocol. [7] presented a simple deadlock detection mechanism caused by circular synchronous group remote procedure calls. In contrast with all these, we limit ourselves to apply finite model-checking techniques to abstract semantic models. Our pNets semantic model is very helpful in this matter, providing us with a very expressive and compact formalism, but where the usage of parameters is limited in a way that can be easily abstracted to finite instances.

Group-based systems as well as parameterized systems are particular infinite systems in the sense that each of their instances are finite but the number of states of the system depends on one or several parameters. Among these parameters we can distinguish: data structures or variables (e.g., queues, counters), number of components involved in the system, ... Automatic verification of such systems has

to face state explosion problem. A variety of techniques to alleviate state explosion has been investigated. We can cite: techniques based on abstraction [26, 16]; techniques based on finding network invariants [20, 32, 15, 30], which can (possibly-over) approximate the system with an infinite family of processes. Others [18, 19] based on finding an appropriate cut-off value of the parameters to bound the system model. For automatic verification of infinite-state systems [13, 3] propose regular model checking. The approach is based on the idea of giving symbolic representation in term of regular languages. Our work tries to take the best of these approaches: whenever possible, we use property-preserving abstractions to build very small (abstract) data domains for the parameters of the basic processes of our systems; but for parameterized topologies such abstractions are not generally complete, so we have to use cut-off strategies as in bounded model-checking.

In the following of the paper, Section 2 overviews *ProActive* communication model and group concepts, and introduces a running example. Section 3 presents our theoretical model and its graphical syntax. Section 4 provides a behavioural model for group communication and synchronisation. Section 5 shows our verification methodology, with experimental results on state-space generation and verification of properties.

2 The *ProActive* communication model

ProActive is an LGPL Java library [2] for parallel, distributed, concurrent applications. It is based on an active object model, where active objects communicate by asynchronous method invocation (called *requests*) with futures: upon a method invocation on an active object, a request is enqueued at the remote object's side, and a future is automatically created to represent the result of the request while the caller continues its execution. Active objects are mono-threaded and treat the incoming invocations one after the other, returning a value for the request at the caller as soon as a request is finished. As remote invocations and future creation are handled transparently, the programmer can write distributed applications in a much similar manner to standard sequential ones. In *ProActive* there is no shared memory between active objects to prevent data race-conditions; consequently, a copy of the request arguments are transmitted to the remote active objects.

2.1 *ProActive* Groups

In this paper, we focus on the group communication mechanism offered by *ProActive* [6]. Groups in *ProActive* work as follows: a group of active objects is a set of active objects that behaves as follows. First, a method invocation on the group results in a remote invocation to all the members of the group in parallel. Second, a list of futures is automatically created to receive the results returned by the group members. Groups are typed as usual objects, and thus invocations to a group are made transparently, as any object invocation. This way, specific primitives for groups are only group creation and management, and thus code modification to handle group communication is minimal. In *ProActive*, groups are dynamic in the sense that objects can join or leave the group at runtime. The main *ProActive* primitives for handling groups are the following:

- *Group ProActive.newActiveGroup*(*String Type*) creates a new group of the type "`Type`".
- **void** *Group.add*(*Object o*) adds an object to a group.
- **void** *Group.remove*(**int** *index*) Remove the object at the specified index.

2.2 Synchronisation for *ProActive* Groups

For classical active objects, synchronisation occurs as follows: a simple access to the future representing the result of a request automatically blocks until the result is computed, and the future is filled. For a group invocation, there is one result by group member, those results are stored in a group of futures. Synchronizing on a group of futures is more complex, here are 3 synchronization primitives of *ProActive*:

- **void** *ProActiveGroup.waitAll*(*Object FutureGroup*) blocks until all the futures of the group return.
- **void** *ProActiveGroup.waitN*(*Object FutureGroup*, **int** *n*) waits until n futures are returned.
- *Object FutureListGroup.waitAndGetTheNth*(*Object FutureGroup*, **int** *n*) waits for the result from the n-th member and returns it.

2.3 Example

To illustrate group communication, we consider an application synchronising meetings, it consists of a master initiator and several clients that contain the agendas of the participants. The initiator suggests a date to all participants that reply whether they are available or not. For this, we define a class *Participant* :

```
public class Participant {
    Booolean suggestDate(Date d) { ... }
    Boolean validate() { ... }
    void cancel() { ... }
}
```

The following code can be implemented by the initiator to coordinate the meeting:

```
public static void main(...) {
....
// group creation
  Participant participants=ProActive.newActiveGroup("Participant");
  ...
// we populate the group by adding one or several element
  participant = ProActive.newActive("Participant",null);
  participants.add(participant);
  ...
  while (true) {
// then we sugggest a date to all members simply by:
  Object answers = participants.suggestDate(date);
  ...
// collateResults gets the result and provides an overall result,
// e.g. returns true if all futures are true
  if (collateResults(answers, ProActiveGroup.size(participants))) {
          Object f=participants.validate(); // validate the meeting
          waitAll(f); // waits until everybody acknowledged validation
          }
  else
          participants.cancel(); // cancel the meeting
    ... }
}
```

This example illustrates well the different mechanisms of group management and communication: first an empty group is created (*newActiveGroup*), then it is populated by several Participant objects.

Thus when the initiator invokes *suggestDate* on the group, this broadcasts a meeting request to all the members. Then the members reply, which fills the futures contained in the group of futures *answers*. The local method *collateResults* synchronises the returns from all these invocations. Validate or cancel is broadcasted to all the group members depending on the result of the preceding step. To illustrate more synchronization mechanisms, the initiator waits until all participants acknowledge the validation. A possible implementation of the *collateResults* method is the following:

```
boolean collateResults(Object ans, int size) {
  boolean result=true;
  for (int i=0 ; i < size ; i++) {
   if (!ProActiveGroup.waitAndGetTheNth(ans,i)) result = false;
 }
 return result;
}
```

Fig. 1 illustrates the mechanism of group communication as implemented in *ProActive*. A method call to a remote activity goes through a proxy, that locally creates "future" objects, while the request goes to the remote request queues.

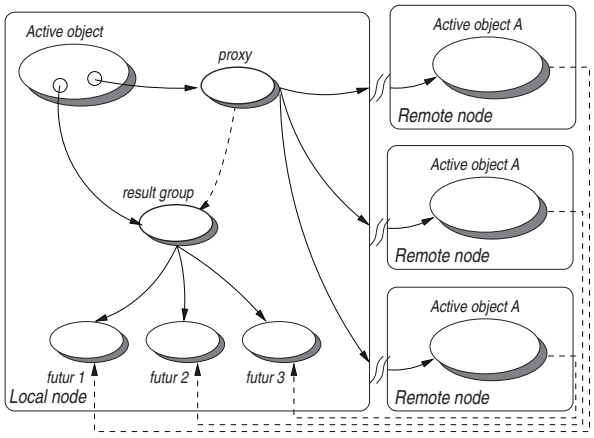


Figure 1: Asynchronous and remote method call on group

3 Theoretical Model

In [8] we have proposed a formalism to represent the behavior of distributed applications. Behavior of complex systems can be represented hierarchically by composition of classical LTSs [29]. Those LTSs are composed using synchronisation Networks (Net) [4, 5] so that the synchronisation product generates a LTS which can be used at the higher level of hierarchy. Finally the behavior of the system can be expressed by a global LTS. We have also shown that this model can be used as an intermediate format to check behavioral properties like temporal ones.

To encode both families of processes and data value passing communication LTSs and Nets are enriched with parameters [14]. Parameters can be used as communication arguments, in state definitions, and in synchronisation operators. This enables compact and generic description of parameterized and

dynamic topologies. In the following we recall definitions of the *parameterized Networks of synchronised automatas (pNets)* as given in [8]. We start by giving the notion of parameterized actions.

Definition 1 Parameterized Actions. *Let P be a set of names, $\mathcal{L}_{A,P}$ a term algebra built over P, including at least a distinguished sort A for actions, and a constant action τ. We call $v \in P$ a parameter, and $a \in \mathcal{L}_{A,P}$ a parameterized action, $\mathcal{B}_{A,P}$ is the set of boolean expressions (guards) over $\mathcal{L}_{A,P}$.*

A describes the possible actions representing interactions between processes. Main actions of our system are illustrated in bold fonts in Figure 2. The typical shape of an action is **!Participant[i].Q_Suggest(f,Date)** for a message **Q_Suggest** sent to the member number **i** of the process family **Participant**. **f** and **Date** are the message parameters, here **f** is the future for the request, and **Date** the request parameter. **!** indicates an emission, and **?** a reception. In most cases the destination of the message can be inferred by the context, and in the figure by the destination of the arrows, in that case, the actions look like **?Q_Cancel()**.

Definition 2 pLTS. *A parameterized LTS is a tuple $\langle P, S, s_0, L, \rightarrow \rangle$ where:*

- *P is a finite set of parameters, from which we construct the term algebra $\mathcal{L}_{A,P}$,*
- *S is a set of states; each state $s \in S$ is associated to a finite indexed set of free variables $fv(s) = \tilde{x}_{J_s} \subseteq P$,*
- *$s_0 \in S$ is the initial state,*
- *L is the set of labels, $\rightarrow$ the transition relation $\rightarrow \subset S \times L \times S$*
- *Labels have the form $l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} \rangle$ such that if $s \xrightarrow{l} s'$, then:*
  - *α is a parameterized action, expressing a combination of inputs $iv(\alpha) \subseteq P$ (defining new variables) and outputs $oe(\alpha)$ (using action expressions),*
  - *$e_b \in \mathcal{B}_{A,P}$ is the* optional *guard,*
  - *the variables $\tilde{x}_{J_{s'}}$ are assigned during the transition by the* optional *expressions $\tilde{e}_{J_{s'}}$*

  with the constraints: $fv(oe(\alpha)) \subseteq iv(\alpha) \cup \tilde{x}_{J_s}$ and $fv(e_b) \cup fv(\tilde{e}_{J_{s'}}) \subseteq iv(\alpha) \cup \tilde{x}_{J_s} \cup \tilde{x}_{J_{s'}}$.

We defined Networks of LTSs called Nets in a form inspired by the *synchronisation vectors* of Arnold and Nivat [4], that we use to synchronise a (potentially infinite) number of processes. The Nets are extended to pNets such that the holes can be indexed by a parameter, to represent (potentially unbounded) families of similar arguments.

Definition 3 *A* **pNet** *is a tuple $\langle P, pA_G, J, \tilde{p}_J, \tilde{O}_J, \overrightarrow{V} \rangle$ where: P is a set of parameters, $pA_G \subset \mathcal{L}_{A,P}$ is its set of (parameterized) external actions, J is a finite set of holes, each hole j being associated with (at most) a parameter $p_j \in P$ and with a sort $O_j \subset \mathcal{L}_{A,P}$. $\overrightarrow{V} = \{\overrightarrow{v}\}$ is a set of synchronisation vectors of the form: $\overrightarrow{v} = \langle a_g, \{\alpha_{t_i}\}_{i \in I, t \in B_i} \rangle$ such that: $I \subseteq J \wedge B_i \subseteq \mathcal{D}om(p_i) \wedge \alpha_{t_i} \in O_i \wedge fv(\alpha_{t_i}) \subseteq P$*

Each hole in the pNet has a parameter p_j, expressing that this "parameterized hole" corresponds to as many actual processes as necessary in a given instantiation of its parameter. In other words, the parameterized holes express *parameterized topologies* of processes synchronised by a given Net. Each parameterized synchronisation vector in the pNet expresses a synchronisation between some instances ($\{t\}_{t \in B_i}$) of some of the pNet holes ($I \subseteq J$). The hole parameters being part of the variables of the action algebra, they can be used in communication and synchronisation between the processes.

Figure 2 gives an illustration of a graphical representation of a parametrized system in our intermediate language. It shows a meeting system with a single initiator and an arbitrary number of participants.

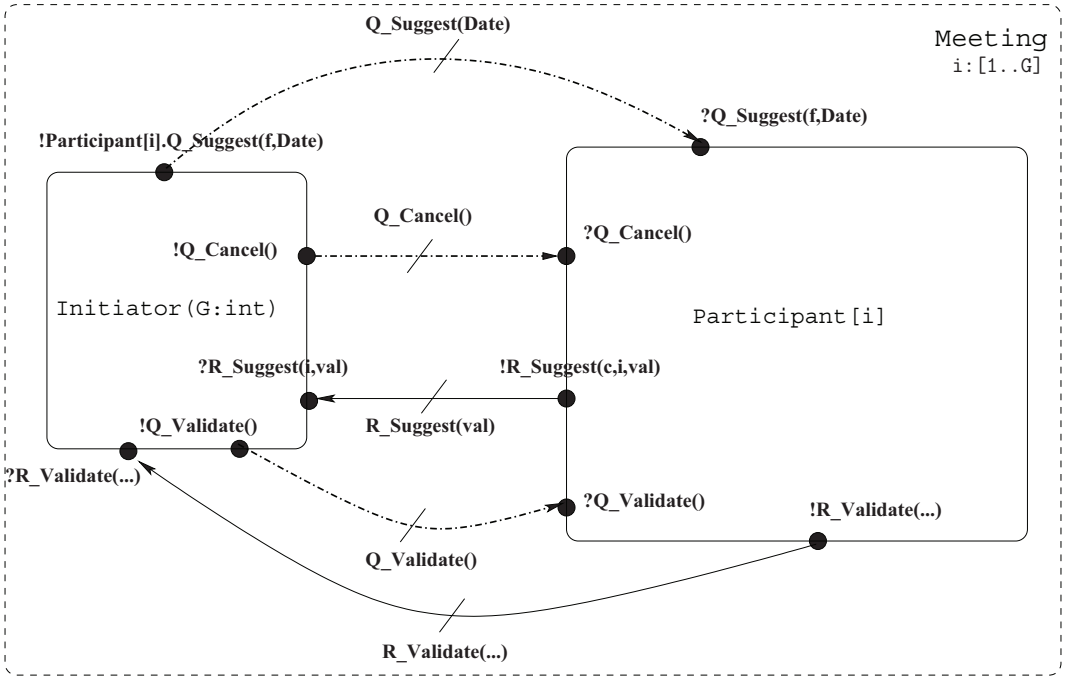


Figure 2: Graphical representation of a parameterized network

The parameterized network is represented by a set of three boxes, INITIATOR and PARTICIPANT boxes inside MEETING box (hierarchy). Each box is surrounded by labelled ports encoding a particular Sort (sort constraint pA_G) of the corresponding pNet. The box will be filled with a pLTS or another pNet (see Fig. 4) satisfying the Sort inclusion condition ($L \subseteq pA_G$). The ports are interconnected through edges for synchronization. Edges are translated to synchronisation vectors. In previous works we only had single edges with simple arrows having one source and one destinations, which were translated into synchronisation vectors of the form `(R_Validate(),!R_Validate(),?R_Validate())` expressing a rendez-vous between actions `!R_Validate()` and `?R_Validate()`, visible as a global action `R_Validate()`. Next section details synchronisation vectors for the multiple arrows we use in our example.

4 Behavioural Model for *ProActive* Groups

In [9] we presented a methodology for generating behavioural model for *ProActive* distributed applications, based on static analysis of the Java/*ProActive* code. This method is composed of two steps: first the source code is analysed by classical compilation techniques, with a special attention to tracking references to remote objects in the code, and identifying remote method calls. This analysis produces a graph including the method call graph and some data-flow information. The second step consists in applying a set of structured operational semantics (SOS) rules to the graph, computing the states and transitions of the behavioural model.

The contribution of this paper is to extend our previous with support for group communication and complex synchronizations related to group communication.

The behavioural model is given as a pNets, which we use as an intermediate language. We express

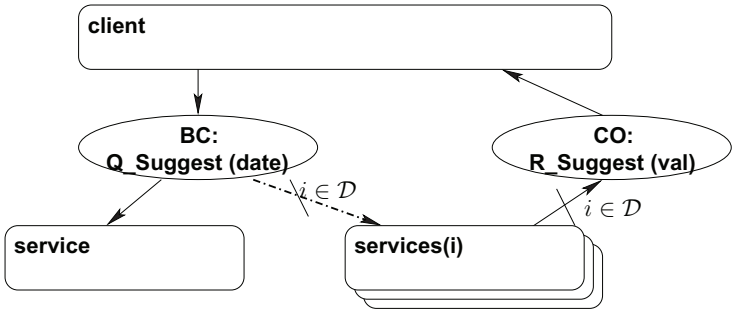


Figure 3: Graphical representation of broadcasting operator

here the semantics of group communication in this intermediate language and show how behaviour of application including group communications with various synchronisation policies can be expressed.

4.1 Modeling complex synchronisations

In order to encode the simultaneity of several message reception/sending, we use a particular kind of proxy and N-ary synchronisation vectors. In Fig. 3 we give a graphical notation for two operators, the ellipse on the left shows a broadcasting operation, and the one on the right show a collection operation.

The first operator that is in charge of broadcasting requests to multiple processes. It is represented by an ellipse with one link arriving from a process, and a set of link departing from the ellipse. The incoming action is triggered as the same time as all the outgoing ones: in the example the output of the client is triggered at the same time as the input in the service on the left, and the input in all the services on the right (the dotted arrow denotes a multiple link). We extend parameterized vectors to support the multicasting communication.

For broadcasting, we introduce the `BC` operator to encode a family of synchronized processes. The vector $< Q\_suggest, !Q\_suggest(date), BC\ i \in \mathcal{D}.?services[i].Q\_suggest(date), ?service.Q\_suggest(date) >$ indicates the synchronisation between one instance of the network 1 (client), a given number of network 2 (services), and another service process. The synchronisation is an observable action labeled $Q\_suggest$. The parameter i ranges in the domain $\mathcal{D}$. For instance, if $\mathcal{D} = [0..1]$, then the vector is expanded to:
$< Q\_suggest, !Q\_suggest(date), ?services[0].Q\_suggest(date), ?services[1].Q\_suggest(date), ?service.Q\_suggest(date) >$.

The operator on the right side collects communications: it synchronizes *one* of its input with its single output. For encoding such a synchronisation, we introduce the `CO` operator to encode a set of synchronisation vectors. The vector $< R\_suggest(val), ?R\_suggest(i, val), COi \in \mathcal{D}.\ !services[i].R\_suggest(val) >$ indicates the synchronisation between a `R_suggest` action in the network 1 (client) and an output of one of the network 2 (services). For instance, with $\mathcal{D} = [0..1]$, this vector is expanded to several vectors:
$< R\_suggest(val), ?R\_suggest(0, val), !services[0].R\_suggest(val), * >$
$< R\_suggest(val), ?R\_suggest(1, val), *, !services[1].R\_suggest(val) >$

Those two synchronisation mechanisms will be further illustrated in the encoding of the example.

4.2 Modeling the Example

We describe now the behavioural model for our example application, especially focusing on the modeling of group proxies, and the communications involving groups. The full model for our example is shown in

Fig. 4. The model is split into two parts interconnected by parameterized synchronization vectors.

- *The initiator* encodes a client side behaviour. The Initiator contains a body encoding an abstraction of the functional code, and the group proxies. For each remote method call in the Initiator code there is a parameterized group proxy, representing an unbounded number of future proxy instances. The body repeatedly suggest a date and either cancel or validate depending on the answers.
- *The participants* encodes the server side behaviour. They are modelled by an indexed family of processes, each representing the behaviour of one element of the group, with its request queue, its body serving requests one after the other in a FIFO order, and the code of its local methods.

A **Proxy** pNet (box) is created for each remote method invocation. The Proxy is indexed by the program point (c) where the method is called. The **Proxy** pNet models the creating and the management of the group of futures: Once the group of future is created, futures can be received one after the other, and each already received future can be accessed. It is also possible to wait until N answers are received.

For each remote method call of the Initiator, a broadcast node, synchronizes the sending of the method call by the initiator body, the initialisation of the corresponding future, and the reception of the request message in the queues of each of the participants in the group.

Concerning the user code, the **Body** boxes in Fig. 4 represent the behaviour of the main method of each active object, again on the form of a pLTS. The code for each method (e.g. **Validate**) is also expressed by a pLTS, and triggered when serving the corresponding request, or by direct invocation like **collateResult**. Each of them is either obtained by source code analysis, or provided by the user.

As it is the only object to act as a server, the participant has a **Queue** box. The corresponding pLTS encodes a FIFO queue of request that is accessed by the participant's body, and filled when the initiator sends a request. The queue can be given a maximum length and raise an error if it is overflowed.

4.3 Variations on group synchronisations

ProActive provides various primitives (see Section 2.2) allowing the programmer to control explicitly the synchronization of asynchronous methods calls by waiting the incoming replies. The network **Proxy_suggest** in Fig. 4 specifies three kinds of these primitives: *waitAll*, *waitN* and *waitAndGetTheNth*. Those three primitives show the different synchronisations that our group proxies can express: counting the number of returned objects, or returning a specific result. They are encoded very naturally using a table of received results, and the number `N` of results already returned. Those information are updated when receiving messages from a *collection* (`CO`) of different results as explained in Section 4.1. Additionally to those primitives, one could also use a *waitOne* primitive waiting for one result, no matter of which it is; this primitive could be encoded with a little more effort by our proxy, but we do not present it because it is not used in our example and we believe it is less crucial than the others. *waitOne* is useful in the case several workers perform the same task, and only one result is necessary.

5 Verification and Results

In principle, the steps for designing and validating a distributed application with our approach are:

1. Specify the structure and the behaviour of the application, in terms of active objects (or components). We provide editors for distributed components in the Vercors platform; specific component interfaces exist for group communication. Alternatively one could imagine tools for static analysis of Java/ProActive code, that would provide a similar abstraction of the system.

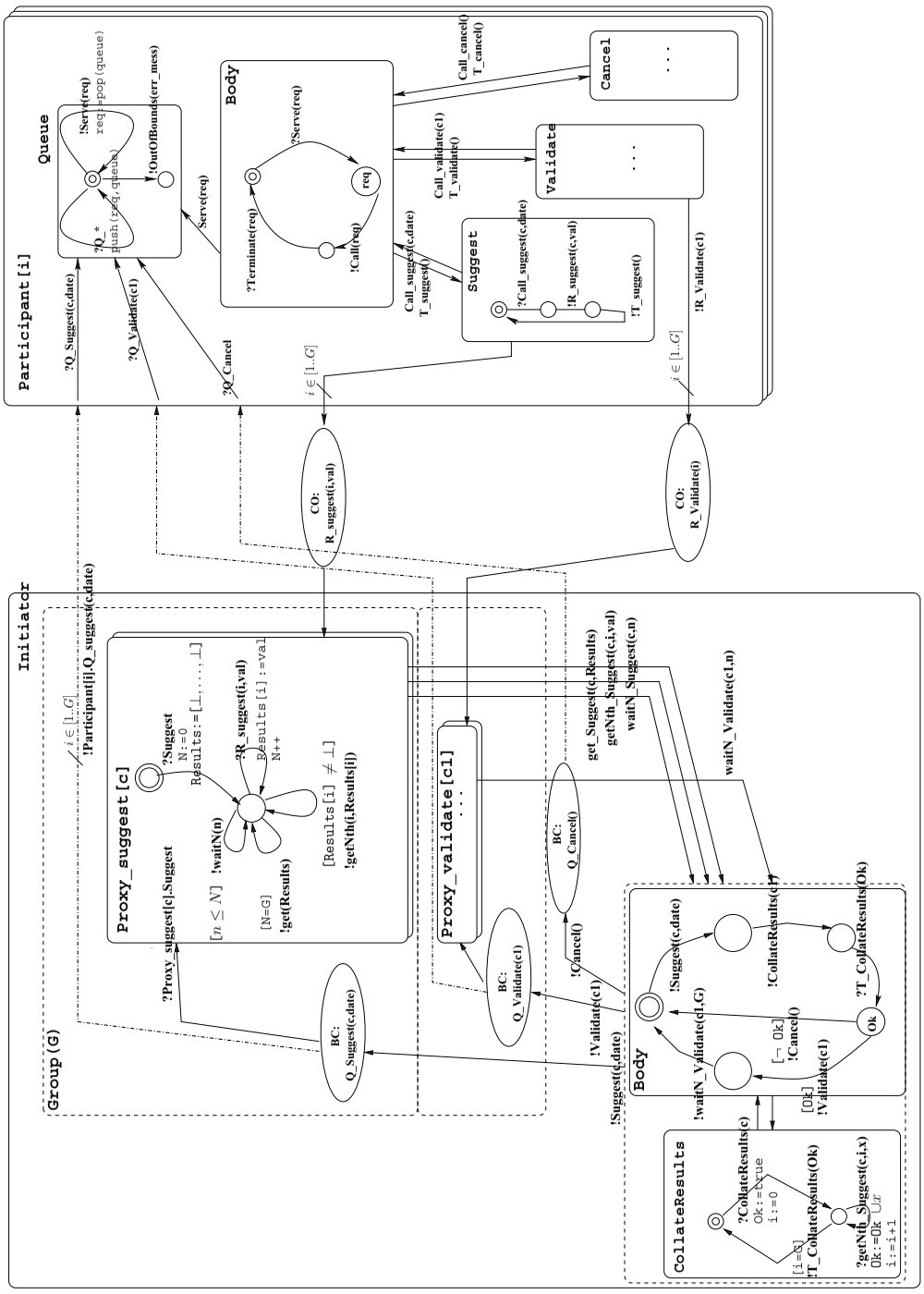


Figure 4: Model of a communication by broadcasting

Abstract data domains: *Group* index: G $\in$ [0..2], *Q_Suggest* argument: $data \in \{D1, D2\}$, *Q_Suggest* result: *bool*
Observed sorts:
Initiator sort: $\{Q\_Suggest(data), Q\_Validate(), Q\_Cancel(), R\_Suggest(index, bool), R\_Validate(index), T\_CollateResults(bool)\}$
Participant sort: $\{Q\_Suggest(data), Q\_Validate(), Q\_Cancel(), R\_Suggest(bool), R\_Validate(), Error()\}$
ParticipantGroup sort: $\{Q\_Suggest(data), Q\_Validate(), Q\_Cancel(), R\_Suggest(index, bool), R\_Validate(index), Error()\}$
System sort: $\{Q\_Suggest(data), Q\_Validate(), Q\_Cancel(), R\_Suggest(index, bool), Error(), T\_CollateResults(bool)\}$

Subsystem	**brute force**		**minimized**		**gen. + min.**
	nb states	**nb transitions**	**nb states**	**nb transitions**	(seconds)
Single Participant	1 801	5 338	90	376	8.2
Initiator	3 163	152 081	54	1 489	11.3
Full system:					
with 3 participants, queue[1]	85 213	839 188	178	489	17.9
with 3 participants, queue[2]	170 349	1 646 368	458	1 284	406.0

With Distributed generation	**generation algorithm**	**Total Time**	**States/Transitions**	**States/Trans (minimized)**
Full system with 3 participants	brute force	6'45"	170 349 / 1 646 368	458 / 1 284
(8x4 cores)	tauconfluence	30'	5591 / 14 236	458 / 1 284
Group of 2 participants	brute force	11'32"	13 327 161 / 48 569 764	4 811 / 24 588
(15x8 cores)	tauconfluence	1150'55"	392 961 / 1 354 948	4 811 / 24 588
Group of 3 participants	tauconfluence	-	*Out of memory*	-
(15x8 cores)			*estimate* $\geq 10^{11}$ *states*	

Figure 5: Size of the generated state spaces for different sub-systems of our example

2. Generate a pNet model, following the approach in the previous section. We plan to have tools automatizing this step in a near future, integrated in the Vercors platform.
3. Write user requirements, in the form of logical formulas in some temporal logic dialect (most action-based logics will be suitable).
4. Use a model-checker to check the validity of theses formulas on the generated model. Currently only finite-state model-checkers are capable to analyse our models. This means that the parameterized pNets have to be instantiated first to a finite system, and that the formulas have to be instantiated accordingly.

The reader acquainted with model-checkers will have guessed that such models are severely exposed to state explosion. It is very important here to observe two facts: First we only work with an abstraction of the system. We use finite abstractions of data-values in the description of data domains, and we only expose (and observe) the events that are useful for the properties. Secondly, we make use as much as possible of the congruence properties of our semantic model: we build the state-space in a hierarchical manner, often minimizing partial models using branching bisimulation before building their products. But this strategy has limits, and sometimes it is better to build the state-space of a subsystem under the constraints of its environment, avoiding unnecessary complexity; this is illustrated in our case-study by the "Participant group" that has by itself a very high state complexity, of which only a small part is used by the "Initiator" client.

In Figure 5 we give figures obtained on our example. The systems in the first 4 lines of the table have been computed on a Fedora 10 box, with 2 dual-core Intel processors at 2.40 GHz, with a total of 3.8 Gbytes of RAM. The source specification was written in the intermediate format Fiacre [12, 11], and the state space generated using CADP version 2008-h. The systems in the last part of the table have been computed on a cluster with 15 nodes, each having 8 cores and 32 Gbytes of RAM. We have been using the Distributor tool of CADP for distributed state-space generation, with or without on-the-fly reduction by tauconfluence [21]; the distributed state space has to be merged into a single state space before minimization and model-checking. The execution times in this part include the deployment of the application, the distributed generation, the merging and the minimization of the resulting state-space. A cell with a "-" means that the computation did not terminate.

The main lesson from this experiment is that intermediate systems will often cause the main bottlenecks in the system construction. Here, an unconstrained model for a group of 3 participants is already too big to be computed on a single desktop machine. By contrast, computing the behavior of such a group in the context of a specific client is feasible (here the model of the full system with 3 participants remains reasonably small). Generating the state-space in a distributed fashion gives us the capability of handling significantly larger models. On-the-fly reduction strategies are useful too, but to a certain point only, because it may involve local computations that require large local memory space themselves. In our tests the generation of the model of a group with 3 participants failed: we estimated that the brute force model has approximately 125 billiards of states (this would require some 12 Terabytes of distributed RAM, 25 times more than our full cluster). But even using on-the-fly reduction by tauconflence, local computations caused an out-of-memory failure.

Proving properties We give here examples of functional behavioural properties that we checked on various scenarios. For this, we have built the global synchronisation product of the system, with 3 Participants in the group (the number of participants does not change the results), and with the size of requests queues instantiated to 1 or 2 depending of the cases.

For expressing the properties, we could use any of the logical languages provided within the CADP tool suite, including LTL, CTL, or specification patterns [17]. In general, we use the regular alternative-free μ-calculus formalism, which is a powerful modal logic, nicely expressing action sequences as regular expressions; it is the native logics of the model-checker. We have checked the following formulas:

1. $< True* .Error > True$: in the system with queue of length 1, the queues can signal an Error.
2. $[True* .Error]False$: in the system with queue of length 2, the queues never signal an Error.
3. $< True* .R\_suggest(i,b) > True$: some paths lead to a response to the `suggest` request.
4. $< True* .T\_CollateResult(false) > True$: the collection of results by the Initiator can return `false`.
5. $After\ !Q\_Suggest(id)\ Eventually\ !Q\_Cancel() \vee !Q\_validate()$: inevitable reachability of either a validation or a cancellation after a date has been suggested. This formula is written in the specification patterns formalism, and expresses correct progress of the system.

Properties 1. and 2. are checked on two different models, with different size of the queue. They prove that a bounded queue of length 2 is required and sufficient to ensure the correct operation of the system. The Error action in the queue of a participant signals that a request is received in a state where the Queue is already full.

Properties 3. and 4. check the reachability of some possible events; technically, property 3 has to be checked for each possible values of parameters i and b, because the μ-calculus logic is not parameterized.

Property 5. expresses the correction of the (first iteration of the) behaviour of the system: in response to a suggest request, we guarantee that the initiator sends either a validation or a cancellation message.

It is interesting to discuss the tools available for exploring and debugging the generated systems. In addition to the model-checking and minimization engines, we have used tools for:

- exploring interactively the generated behaviour at the level of its Lotos representation (OCIS)
- displaying graphically the generated LTS (BCG_EDIT)

Consider formula 1 that checks reachability of action Error. In addition to a "True" result, the model-checker produces a trace illustrating the reachability from the initial state, as shown in Figure 6. The trace consists in a full cycle through the system behaviour, from the initial state to state 6 and action "Q_cancel()". Then, because we do not wait for the return of the Cancel requests, one of the Participants can still have a Cancel request pending in its queue when the Initiator sends the next Suggest request, which leads to an Error. The BCG_EDIT tool can display the sequence of Figure 6. A finer trace showing internal interactions and allowing user-driven guidance of the system can be obtained with the OCIS tool.

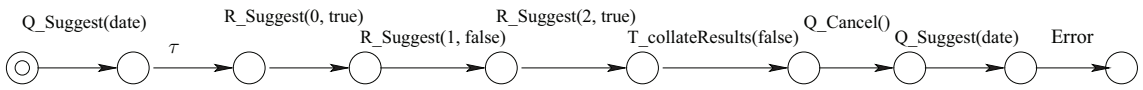


Figure 6: Path containing the Error action

6 Conclusion

In this paper we have sketched models for specifying and verifying the correct behaviour of group-based applications. Our parameterized models enable the finite representation of groups of arbitrary size, and express the communication with such groups, together with the associated synchronizations. For our modelling, we focused on the *ProActive* library; nevertheless these models can be applied to other middlewares involving collective communications. Our parameterized models are supported by model checking tool. Besides they are hierarchical labelled transition systems, therefore suitable for analysis with verification tools based on bisimulation semantics.

Our main contribution is to provide a behavioural semantic model for group communication applications. It allows the application programmer to prove the correctness of his/her behavioral properties, and for instance detect deadlocks [7]. We have illustrated our approach on an example application, generated the corresponding model, and proved several properties ensuring the correct behaviour of the example. The size of the generated system and the proven properties show that, if the system is entirely known at instantiation time, we are able to prove non-trivial properties on examples of a reasonable size.

Towards dynamic groups A nice perspective of this work is the verification of groups with dynamic membership. The *ProActive* middleware allow active objects to join and leave a group during execution. This way the application can adapt dynamically in the case new group members are necessary to perform a complex computation, or systematically when new machines join the network. The use of pNets will facilitate the specification of dynamic groups thanks to the support for parameterized processes and synchronisation vectors.

References

[1] *JGroups - A Toolkit for Reliable Multicast Communication*. Http://www.jgroups.org/index.html.

[2] *ProActive - Programming, Composing, Deploying on the Grid*. Http://proactive.inria.fr/.

[3] P. A. Abdulla, G. Delzanno, N. Ben Henda & A. Rezine (2007): *Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems)*. In: *TACAS*, pp. 721–736.

[4] A. Arnold (1994): *Finite transition systems. Semantics of communicating sytems*. Prentice-Hall. ISBN 0-13-092990-5.

[5] A. Arnold (2002): *Nivat's processes and their synchronization*. *Theor. Comput. Sci.* 281(1-2), pp. 31–36.

[6] L. Baduel, F. Baude & D. Caromel (2007): *Asynchronous Typed Object Groups for Grid Programming*. *International Journal of Parallel Programming* 35(6), pp. 573–614.

[7] B. Ban: *A Simple Deadlock Resolution Scheme for Synchronous Reliable Group RPC (draft)*. Http://www.jgroups.org/javagroupsnew/docs/papers.html.

[8] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio & E. Madelaine (2009): *Behavioural models for distributed Fractal components*. *Annals of Tlcommunications* 64(1-2), pp. 25–43.

[9] T. Barros, R. Boulifa & E. Madelaine (2004): *Parameterized Models for Distributed Java Objects*. In: *International Conference on Formal Techniques for Networked and Distributed Systems FORTE'04*. LNCS 3235.

[10] J.A. Bergstra, A. Ponse & S.A. Smolka (2001): *Handbook of Process Algebra*. North-Holland. ISBN 0-444-82830-3.

[11] B. Berthomieu, J.P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang & F. Vernadat (2008): *Fiacre: an Intermediate Language for Model Verification in the Topcased Environment*. In: *ERTS 2008*, Toulouse France.

[12] B. Berthomieu, J.P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker & F. Vernadat (Mai 2007): *The syntax and semantics of Fiacre*. In: *Rapport LAAS N07264 Rapport de Contrat Projet ANR05RNTL03101 OpenEmbeDD*.

[13] A. Bouajjani, B. Jonsson, M. N. & T. Touili (2000): *Regular Model Checking*. In: *CAV*, pp. 403–418.

[14] A. Cansado & E. Madelaine (2008): *Specification and Verification for Grid Component-Based Applications: From Models to Tools*. In: *FMCO*, pp. 180–203.

[15] E. M. Clarke, O. Grumberg & S. Jha (1997): *Verifying parameterized networks*. *ACM Trans. Program. Lang. Syst.* 19(5), pp. 726–750.

[16] E. M. Clarke, M. Talupur & H. Veith (2006): *Environment Abstraction for Parameterized Verification*. In: *VMCAI*, pp. 126–141.

[17] Matthew Dwyer, George S. Avrunin & James C. Corbett (1998): *Property Specification Patterns for Finite-State Verification*. In: *Proceedings of the Second Workshop on Formal Methods in Software Practice*, ACM Press, pp. 7–15.

[18] E. A. Emerson & K. S. Namjoshi (1995): *Reasoning about rings*. In: *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 85–94.

[19] E. A. Emerson, R. J. Trefler & T. Wahl (2006): *Reducing Model Checking of the Few to the One*. In: *8th international conference on formal engineering methods, ICFE*, pp. 94–113.

[20] E.A. Emerson & K.S. Namjoshi (1996): *Automatic verification of parameterized synchronous systems*. In: *Information Processing Letters*, 8th International Conference on Computer Aided Verification, CAV'96, Rutgers. 22(6):307-309.

[21] H. Garavel & G. Serwe (2006): *State space reduction for process algebra specifications*. *Theorical Computer Science* 351(2).

[22] Alan J. Hu, Rui Li, Xizheng Shi & Son T. Vuong (1999): *Model-Checking a Secure Gorup Communication Protocol: A Case Study*. In: *FORTE*, pp. 469–478.

[23] A. Kupšys, S. Pleisch, A. Schiper & M. Wiesmann (2004): *Towards JMS compliant group communication - a semantic mapping.* In: *Proceedings of the 3rd International Symposium on Network Computing and Applications (IEEE NCA04)*, IEEE, Cambridge, MA, USA.

[24] M. Kwiatkowska & G. Norman (2002): *Verifying Randomized Byzantine Agreement*. In: *FORTE (LNCS 2529)*, Springer Berlin / Heidelberg, pp. 194–209.

[25] M. Layouni, J. Hooman & S. Tahar (2003): *On the Correctness of an Intrusion-Tolerant Group Communication Protocol.* In: *CHARME*, pp. 231–246.

[26] D. Lesens & H. Saïdi (1997): *Abstraction of parameterized networks*. *Electr. Notes Theor. Comput. Sci.* 9.

[27] M. Li & M. Baker (2005): *The Grid: Core Technologies*. Wiley. ISBN: 0-470-09417-6.

[28] M. Massink, J-P. Katoen & D. Latella (2004): *Model Checking Dependability Attributes of Wireless Group Communication*. In: *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, IEEE Computer Society, Washington, DC, USA, p. 711.

[29] Robin Milner (1989): *Communication and Concurrency*. Prentice Hall. ISBN 0-13-114984-9.

[30] A. Pnueli & E. Shahar (2000): *Liveness and Acceleration in Parameterized Verification*. In: *CAV*, pp. 328–343.

[31] A. Schiper (2006): *Dynamic group communication*. *Distributed Computing* 18(5), pp. 359–374.

[32] A. Prasad Sistla & V. Gyuris (1999): *Parameterized Verification of Linear Networks using Automata as Invariants*. *Formal Asp. Comput.* 11(4), pp. 402–425.

Experiments with distributed Model-Checking of group-based applications

Ludovic Henrio & Éric Madelaine
INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France
Email: First.Last@sophia.inria.fr

I. Behavioural models for Group-based applications

In recent work [3], we have proposed a modelisation of the behaviour of group-based distributed applications, in the form of parameterized networks of synchronised automata (pNets, see [4]). A typical structure in group-based applications is illustrated in Figure 1, where a client sends requests using a synchronous broadcast mechanism (BO) to a number of servers, then collects (CO) the results from these requests in an asynchronous way.

The pNets formalism provides us with a powerful and flexible way to encode labelled transition systems with value-passing, as well as parameterized topologies of processes, and many different communication primitives. But it also has the great advantage that it can be transformed by abstraction into finite pNet models, suitable for finite-state model-checking.

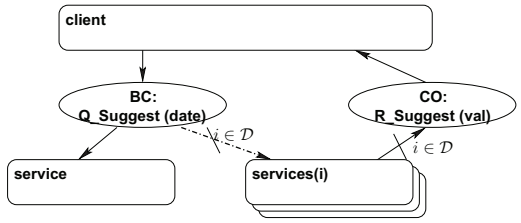


Figure 1. Graphical representation of broadcasting operator

II. Encoding with the Fiacre intermediate language

The middle term goal of these experiments is to integrate automatic model-generation procedures in our VerCors toolset [6]. VerCors includes (graphical) editors for the definition of distributed component-based applications; from such a description, the system generates a pNet behaviour model, that needs to be translated into a language usable as input of a model-checker. We use the CADP verification toolset [9]. Amongst the possible input languages for the CADP engines, we have chosen the recently defined Fiacre format [5], featuring most of the concepts we need for encoding our pNet structures: simple constructive data-types, automata-like processes, parameterized processes, multi-process communication (a la Lotos).

In Figure 2 we show the high-level architecture of our case-study: *Participant[i]* is a group of processes providing services *Suggest* and *Validate*. *Initiator* is a client, that may send requests to the whole group in a broadcast manner. Results from these requests are returned asynchrounously by each group member, collected by a proxy, before being used by the Initiator body. Each Participant has a queue, storing the incoming requests; the participant body is monothreaded, and encodes the service policy.

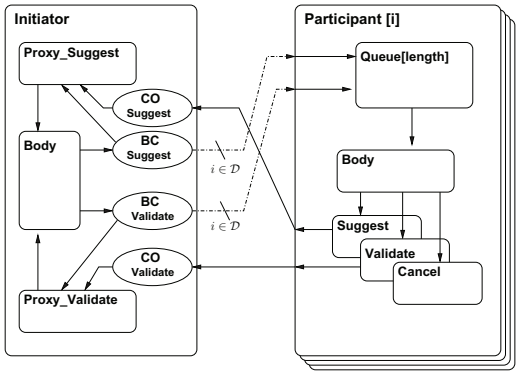


Figure 2. Structure of our case-study

In the current state of our research, we encode manually the pNets into Fiacre code. The most interesting features of the Fiacre language are described here:

Fiacre processes feature standard state-oriented and guarded events concepts, e.g.:

```
process Queue2 [ Q_Suggest: in data,
    Q_Validate: in data2, Q_Cancel:
    none, ... ]
is
states S_empty, S1, S2, ...
var x:data,  y:data2, ...

from S_empty
  select
    Q_Suggest?x; to S1
  []
    Q_Validate?y; to S2
...
end
```

It also support user-defined data types, and classical programming constructs, as in :

```
const G:nat is 3
type fut_data is union undef | b of
    bool end
type Result_vector is array G of
    fut_data
process Group_proxy [ WaitFor_m: none,
    GetNth_m: out indexG#bool, ... ]
is
states A
var val:bool,
    V : Result_vector
from A
       case V[0] of
  undef -> WaitForNth_m ! 0
       | b(val) -> GetNth_m ! 0,val
       end case;
       to A
```

Fiacre components are used to compose processes hierarchically, with parallel operators inspired from Extended Lotos. They feature explicit declaration of ports, and constructs for specifying multi-way synchronisation of events on these ports:

```
component System [Q_Suggest: data, ...]
is
port  R_Validate0, ...: indexG

par Q_Suggest, ... in
   R_Suggest0, R_Validate0, ... ->
   Initiator [Q_Suggest, R_Suggest0,
       R_Validate0, ...]
 ||
   R_Suggest0, R_Validate0
   -> Participant0 [Q_Suggest,
       R_Suggest0,... ]
 ||
   ...
end
```

In this composition we can observe all synchronisation modes useful for the encoding of our pNets: Events on port `R_Suggest0` are synchronized between components `Initiator` and `Participant0`, events on port `Q_Suggest` are synchronised between *all* participating components (our broadcast communications), events on the local port `R_Validate0` are hidden from outside System, while those on `Q_Suggest` are visible in the global system. Components can have parameters, however they cannot encode directly parameterized topologies, because this would require to specify parameterized synchronisation on their ports. For example here we had to declare one separate port `R_Suggest_i` for each of the possible message from `Participant[i]` to `Initiator`.

III. Using distributor on a cluster infrastructure

In the following tables, we show the figures obtained with the distributed version of the CADP state-generation tools. These figures have been obtained on a cluster, comprising 15 nodes; each node has 8 cores and 32 Go of RAM. The table in Figure 3 measures the overhead due to the deployment of the distributed model-checker. The cost is linear in the number of cores, and mainly due to the copy of the engine executable file on all nodes. There is also a quasi-constant cost, due to preliminary compilation of state-generation code, and to final merging and minimization of the generated state-space.

Subsystem	configuration	Total Time
Initiator :	sequential	11”
	3x4 cores	24”
	3x8 cores	33”
	8x4 cores	38”
	15x4 cores	52”
	15x8 cores	89”

Figure 3. benchs for a small component

Figure 4 shows results obtained for bigger systems. The principal source of state explososion in our example comes from the Queue process as it encodes all possible values of requests with their data arguments, in each position of the queue. We encode a bounded queue structure, with a specific OOB event allowing for checking boundedness properties. Playing with the size of data domains, as well as with the group size, is an easy way to experiment with various strategies and resource configurations.

An important remark is that building systems in a pure compositional way is not always the best strategy: the full state-space of subsystems, when computed out of their context, can be much larger that the part really useful. Here we can observe that the full system size is much smaller than the size of the group of participants, and that trying to compute the group state-space by itself may even fail. This of course is not new, and usual solutions include:

1) generate directly the state space of the server(s) together with their client(s). This is what we have done here in the rows for the “Full system”.
2) generate separately the state-space of the server(s), but providing some constraints on the context behaviour. This would be the idea of a “contract” for using the server(s) in a correct manner. In the CADP toolbox, the projector tool is providing this possibility; the context can be computed from the client code, or can be guessed by the server developper, and checked correct later
3) generate separately the state-space of the server(s), and reduce it by (branching) bisimulation before computing any product.

Solutions 2) and 3) technically involve using Fiacre code for describing the individual subsystems, then using the script language of CADP, SVL, to perform the reduction and parallel product operations. This would have been too complicated for our ongoing experiments, and we have concentrated on the pure distributed features of the tools.

The *distributor* tool of CADP generates state spaces in a distributed way, based on a static hash function ensuring the distribution of states on a number of nodes. The resulting states must then be merged before application of tools that are only available in a sequential implementation, including bisimulation-based minimization, and model-checking. However, some partial-order reduction techniques are available "on-the-fly", during distributed state generation, namely taucompression and tauconfluence [8]. They provide a trade-off be-

Subsystem	generation algorithm	Total Time	States/Transitions	States/Transitions (minimized)
Initiator (sequential)	brute force	12”	3 163 / 152 081	54 / 1 489
Initiator (3x4 cores)	brute force	24”	3 163 / 152 081	54 / 1 489
	taucompression	30”	3 163 / 131 942	54 / 1 489
	tauconfluence	35”	1 219 / 33 815	54 / 1 489
Full system with 3 participants (8x4 cores)	brute force	6’45”	170 349 / 1 646 368	458 / 1 284
	taucompression	11’48”	170 349 / 607 570	458 / 1 284
	tauconfluence	30’	5591 / 14 236	458 / 1 284
Single Participant (sequential)	brute force	9’	9 653 / 31 480	171 / 641
Group of 2 participants (15x8 cores)	brute force	11’32"	13 327 161 / 48 569 764	4 811 / 24 588
	taucompression	30’59"	13 327 161 / 48 569 764	4 811 / 24 588
	tauconfluence	1150’55”	392 961 / 1 354 948	4 811 / 24 588
Group of 3 participants (15x8 cores)	tauconfluence	-	*Out of memory*	-

Figure 4. Benchs for the various on-the-fly reduction strategies

tween space and time consumption, generating less transitions and less states at the price of local "on-the-fly" computations. This trade-off is clearly visible on the full system computation figures, where we generate only 5K states in tauconfluence mode (before merging and minimization). Taucompression is significantly less expensive in time than tauconfluence; here, it appeared that it does not give any benefit for the “Groups of participants” cases. Tauconfluence brings significant reduction in the number of generated states, but it appears here that local computation was too costly in local memory space for the “group of 3” case, preventing us to get a measurable result.

IV. Conclusion

Group-based distributed systems are specific cases of distributed applications with a parameterized topology. They are naturally modelled by systems with a very large state-space. We encode the behavioural semantics of group-based applications using the intermediate format FIACRE. We have experimented with model-checking of such systems, using the CADP verification toolset, and in particular the distributor tool. This allowed us to generate very large but finite state-space on the PacaGrid cloud infrastructure. We have then been able to compare different techniques for generating state-spaces, and experiment with different sizes of the modelled system and of the experimental platform.

In practice, an efficient solution would rely on a combination of the techniques mentionned in this paper, and in particular on the use of, at the same time, on-the-fly, hierarchical, and contextual techniques. In particular the last technique allows the partial specification of the context in which the system will be used, which will greatly reduce the state space to be generated, and seems a promising method that we want to experiment in future works.

There exists other implementations of distributed model-checking tools, in particular the DiViNe toolset [1], that implements model-checking algorithm for LTL, with specific optimisation for various computing infrastructures, namely clusters, multi-core, and Cuda; and LTSmin [2], that is a MPI-based tool implementing equivalence checking and minimization for various formalisms. Comparisons between these systems is not easy, as it involves encoding the case-studies in quite different formalisms, e.g. the DVE specification language for DiVinE, or μCRL for LTSmin. Furthermore, these toolsets also implement their model-checking algorithms in a distributed way, while the current version of CADP only supports state-generation and on-the-fly reduction in a distributed way, while minimization and model-checking remain sequential.

As a consequence, a significant comparison should include non-trivial efforts, in each of the systems, to find the best encodings of our semantics into the system’s input format(s), and to find the optimal strategy for combining the state-generation / minimization / model-checking primitives of the various tools. Last, the result of such a comparison will heavily depend on the physical resources available; for example the distributed state-space generation in CADP is specifically dedicated to cluster architecture, and will not take benefit of multi-core or CUDA optimizations.

References

[1] DiVinE, Distributed and Parallel Verification Environment.
[2] LTSmin: Minimization and Instantiation of LAbelled Transition Systems. http://fmt.cs.utwente.nl/tools/ltsmin/.
[3] R. Ameur-Boulifa, L. Henrio, and É. Madelaine. Behavioural models for group communications. In *in proceedings of the International Workshop on Component and Service Interoperability, WICS’10*, Malaga, june 2010. to appear.
[4] T. Barros, R. Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, 64(1–2), jan 2009. also Research Report INRIA RR-6491.
[5] B. Berthomieu, J. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. The syntax and semantics of Fiacre. In *Rapport LAAS #07264 Rapport de Contrat Projet ANR05RNTL03101 OpenEmbeDD*, Mai 2007.
[6] A. Cansado and E. Madelaine. Specification and verification for grid component-based applications: from models to tools. In F. S. de Boer, M. M. Bonsangue, and E. Madelaine, editors, *FMCO 2008*, number 5751 in LNCS, pages 180–203, Berlin Heidelberg, 2009. Springer-Verlag.
[7] A. Cansado, E. Madelaine, and P. Valenzuela. VCE: A Graphical Tool for Architectural Definitions of GCM Components. 5th workshop on Formal Aspects of Component Systems (FACS’08), Sep 2008.
[8] H. Garavel and G. Serwe. State space reduction for process algebra specifications. *Theorical Computer Science*, 351(2), 2006.
[9] F. Lang, H. Garavel, and R. Mateescu. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *CAV’07*, 2007.

Chapitre 8

Conclusion and Perspectives

We have described our development of the pNets behavioural model, and of a specification and verification environment for distributed component-based applications, based on the pNets model. We also have tried to show how these research directions were natural developments of our previous results in the area of process algebras and behavioural semantics.

Our main contributions are :

- On the theoretical side, we have defined a comprehensive, compact, and flexible model called pNets. We have used this model to define the behavioural semantic of many of the main features of our distributed components, including asynchronous remote requests, request queues with user-defined service policy, non-functional controllers (life-cycle, binding controller, attribute controller), future proxies and first-class future management, collective interfaces with group proxies.
- On the practical side, we have developed a set of prototype tools for program specification and model generation, and shown how this approach can be used for non-trivial use-cases, in conjunction with state-of-the-art model-checking and equivalence-checking tools. We have experimented distributed model-checking tools that allows us to scale up to large state spaces (billions of explicite states), built in a hierarchical and compositional manner.

This work was backed-up with 3 PhD thesis and more than 16 studentships, and gave birth to the publication of a journal article, a book chapter, and 14 papers in international conferences.

From this point we develop in this chapter our perspectives for short-term and longer-term research, along 4 different axes :

1. Integration of our prototypes in a verification platform, providing an environment usable by non-specialist developers.
2. Addition of model-checking techniques and engines beyond the current limits of finite (explicit or implicit) state-space representations.
3. Generation of “safe by construction” code.
4. Extension of our models to handle new types of distributed systems.

Platform : Our existing prototypes have to be complemented and tightly integrated, before being usable by non-specialists. In particular we have mentioned in Chapter 5 that we need to choose formalisms, and some related tools, for the specification of automata (FSMs or StateCharts) describing the behaviour of the basic processes of our systems, and for the expression of the logical formulas defining the system requirements. Then the integration of tools will include (see Fig. 5.3) :

- Extension of our ADL2N tool to handle abstraction definition in a coherent way for all specification formalisms (behaviour, architecture, and properties). The tool should give as much help as possible to the user by using static analysis information (method call graphs, data flow analysis, aliases analysis, etc.).
- Generation of pNets structures from the behaviour and architecture formalisms. At this level, it may be useful to consider generating a specific model for each property in the requirement set, allowing for optimal abstraction and for optimal model generation strategies.
- Generation of input files for specific verification toolsets (E.g. Fiacre or LotosNT, plus SVL and EXP files, in the CADP case), implementing verification strategies, and combining all possible state-space reduction techniques in an automatized manner (hierarchical minimization, optimal hiding, contextual behaviour, distributed processing, etc.).
- Translation of verification diagnoses back to user-level formalisms, and display of such diagnoses (transitions and states, traces, or more complex games) within the specification editors.
- Integration of all these elements as plugins within the VerCors eclipse environment.

Beyond Finite-state MC : Even when combining all available techniques for reducing the state-space explosion, it is clear that the processing of large distributed applications may easily exceed the capacity of our verification tools. We foresee several independent ways to overcome current limitations :

First by a more elaborated use of our knowledge of the semantics of data values. In the current approach, we define abstract data domains by identifying values that have an equivalent impact, statically, on the whole system behaviour. This method can (and should) be enhanced by adapting automatically the abstraction to the data occurring in the formulas one wants to prove. But going further, data information can be used to reduce the subset of the transition system that needs to be explored by the model-checker : there is a quite active research domain in the area of symmetry reduction techniques, that exploit symmetry in the structure of processes (when the system includes sets of identical processes), or symmetry in the data manipulated by processes (e.g. in structured data like vectors or sets). These techniques may be a good approach to reduce drastically the size of state-space that is sufficient to represent faithfully the whole system behaviour.

Secondly we should seek to enhance significantly the abstraction level of our models, by using general properties of our specification and programming languages to hide lower levels of details in the construction of behavioural models. We are already using this approach for encoding the semantics of remote requests mechanisms in ProActive/GCM, where generic results have been proved by theorem proving techniques [34, 57], allowing us to encode remote request deposits and result retrievals atomically in the behavioural model. It should be possible to use similar links between theorem-proving and model-checking to prove other properties, more or less specific to some application structures or skeletons, and reduce accordingly the level of detail of the pNets behaviour model.

Last we shall continue looking for specific representations of infinite-state systems where (semi-) decidable satisfiability algorithms exist, as we have started experimenting in the case of unbounded fifo channels. The challenge then is to find a way for combining such representations and algorithms together, without encoding all of them into one single universal model (as such an approach would

loose any advantage of each specific algorithm).

Code Generation : We have mentioned in Chapter 6 the idea of generating code from our specification formalisms, that would be "safe by construction" with respect to the properties proved during the verification phase. While this is a well established approach in the area of hardware, where synthesis methods and tools are widely used, it is not so common in the software development area. Still this is not far from the the idea of Model-driven Development (MDE). The difference is that we do not intend to define specification formalisms precise enough to generate 100% of the final implementation, as this would prevent any effective verification activity.

What we plan is to have specifications that will be as abstract as possible, while containing enough information to enable the full generation of the control part of our components. More precisely, for ProActive/GCM applications, the code of the "runactivity" method of primitive components (which defines the selection policy of incoming requests), and the code for the management of non-functional interfaces of both primitive and composite components can be fully generated, while for the "functional code" of service and local methods the tool can simply generate a skeleton, that will be complemented by the developer.

Then we can provide rules about what can be safely modified or not (e.g. can one change, or add, parameters to a method call of a client interface of the component ?), or even provide assertions to be checked by run-time verification code. An important open question is how to allow (and to help) the developer to define mappings between its own data classes and the abstract (simple) types he has been using in the specification formalisms. Ideas for defining such mappings in a safe way have been experimented e.g. in the Bandera framework [41].

New Challenges : The foreseen architecture of VerCors, and the perspectives listed in the previous paragraphs, should allow us to master the modeling and verification of today's distributed applications, including large scale and dynamically reconfigurable Internet services. But of course technological progress goes fast, and the Moore law applies also to the complexity of software systems. While we can do our best to advocate and teach development methods that will include formal methods, rigorous specification, well-structured development of software, we also face deep changes in the technology that bring intrinsic complexity in computer science.

Multicore is certainly the best example of such a break in technology that brings in, simultaneously, a huge advance in computing power, but also significant difficulties in system and application programming. For us it means a deep reformulation of the basic model of distributed objects, introducing concurrency inside local behaviour of components, challenging the core semantics of our sequential processes. How can we redefine this model to handle the local structure of new processor architectures, allowing for optimization of local performance, while keeping the fundamental semantic properties of the whole model ? The challenge is to handle properly and safely these new development contexts in which programmers are faced with new dimensions of heterogeneity, from multicore architectures, GPU computation units, virtual machines, to dynamic and adaptable software services ready for elastic cloud computing, and Internet-wide applications.

Our next PhD subjects : In the short term, in parallel with the implementation of the most urgent missing tools of VerCors, we have some PhD subjects that are just starting, or still open. This includes :

- **Novel verification methods for distributed software (open)**. This subject aims at extending our approaches beyond the usual frame of finite-state systems in two complementary directions. First, identify finite abstractions of parameterized systems that preserve some useful sets of properties. Second, use dedicated infinite-state (semi) decision procedures able to deal with specific classes of systems, for example infinite communication channels or arithmetic counters,
- **Reliable Integrated and Autonomic Deployment and Management for SaaS composite applications (starting)**. This subject is in the context of an industrial research contrat of the Oasis team, in which we develop GCM-based tools for the management of highly dynamic component structures. These developments will be accompanied by the development of validation methods combining theorem-proving approaches with model-checking of parameterized and dynamic pNets structures.
- **Formal Model and Scheduling Algorithm for Real-time CPS (started)**. This is in the context of a collaborative research with ECNU University in Shanghai (and the Aoste team at INRIA SophiaAntipolis), in which we want to draw bridges between our respective synchronous and asynchronous behaviour models and tools. The (chinese) PhD student will work on abstractions enabling the application of our methods in the context of CPS (Cyber-Physical Systems) with time constraints. This is in some sense a frontier case of our "new challenges" domains.
- **Safe Code Generation (open)**. This is a work that builds up on the approach where the application modeling and verification is done on early abstract formalisms (behaviour and architecture). Then we aim at producing code skeletons (eventually associated with runtime assertions) that will guarantee that the implementation satisfies the properties proved at model level. This work is in the context of our collaboration with the new CIRIC Center in Santiago de Chili.

Chapitre 9

Annexes

9.1 Diplomas

Engineer from Ecole Polytechnique de Paris, 1980

DEA Theoretical Computer Science, University of Paris 7, 1981

PhD (doctorat de 3^{eme} cycle), University of Paris 7, "Système d'aide à la preuve de compilateurs", 1983

9.2 Professional activities

1980-1983 PhD, INRIA project-team "Languages and Translators", adviser M. Nivat & M.C. Gaudel.

1983-1996 Research Scientist (Chargé de Recherche) INRIA, project-team "MEIJE", Sophia-Antipolis.

jan-aug 1992 NSF-INRIA scientist exchange, 7 months visit, Un. of North-Carolina, Raleigh, USA.

1996-2000 Computer systems deputy (Responsable des Moyens Informatiques), INRIA Research Unit of Sophia-Antipolis.

2001-feb.2011 Senior Research Scientist, INRIA project-team OASIS (Active Objects, Semantics, Internet, and Security), Sophia-Antipolis.

since march 2011 Head of the OASIS project-team.

Student directions 4 PhD thesis, 1 Postdoc, 16 master, or engineer internships.

Teaching Courses in Software Engineering, Functional Languages, Reflexive Languages, Operational Semantics, Formal Methods, Verification.
Schools : CERICS, CERISI (DESS, DEA), Telecom Paris (3rd year), University of Nice Sophia Antipolis (Licence, Master 1 & 2)

9.3 Research community responsibilities

Steering Committee of International Conferences : FACS (2006-2011), FMCO (2007-2011)

Program Committee of International Conferences : LDTA'05, Provecs'07, SCCC'07, SAVCBS'08, Qoas'08, SERA'08, FACS (2006-2011), FMCO (2007-2011) , Euromicro-SEAA'09, PDMC'09, FESCA'09, ICPP'11

Review for journals : SCP, IET-Software, l'Objet

Standardization : I have been member, and voting representative for INRIA, from 2008 to today, of the TC-GRID (Technical Committee on Grid Technologies) at ETSI, now renamed as TC-CLOUD. In this standardization body, I have participated, and actively pushed, the creation and approval process for the Grid Component Model GCM. This led to the publication of 5 standards, between 2008 and 2010.

EC Referee : I have acted as a referee for the European commission, in 2007-2008, for the review of FP6 projects.

9.4 Scientific collaboration, projects, contracts

I have participated at different levels to a long series of collaborative projects, in the context of the Meije and Oasis teams. I give below a list of these projects, then I give a more complete description of the Fiacre and ReSeCo projects, that I have coordinated, as well as the ANR international project MCorePhP, that is the most significant of the current Oasis collaborations in terms of verification.

International : NSF-INRIA (1992, with a 7 month research visit at NCSU, Raleigh, USA), Associated Team Oscar (2004-2006, with Universidad de Chili, Santiago, participant then coordinator), Stic-Amsud ReSeCo (2006-2009, Chili, Uruguay, Argentina, coordinator), Stic-Asie Grids (2008-2010, Pakistan, China, participant).

Europe : Lotosphere (1989-1992, Esprit IP, task leader), Concur (Esprit BRA 1989-1990, resp. INRIA), Concur2 (Esprit Bra, 1991-1992, task leader), CoreGrid (FP6 NoE, 2005-2009, participant), GridComp (FP6 Strep, 2006-2009, participant), NessiGrid (FP6 SSA, 2006-2008, INRIA representative).

France : ACI Fiacre (ACI Sécurité, 2005-2007, coordinator), ANR MCorePhP (ANR Blanc International, avec Un. Tsinghua Pekin, 2010-2012, participant).

The FIACRE project

Type and Dates : French ACI Sécurité, sep. 2005 - sep. 2007.
Title : Models and Tools for Safety and Security Analysis of Distributed Components and their Composition
Partners : INRIA Rhône-Alpes EPI Vasy, Feria- IRIT/LAAS, GET-ENST Paris
My role : Creation, Coordination
Abstract

This project was launched with the ambition of strengthening the impact of distributed component based programming on software development methods. In order for this approach to fully work, while component libraries become available, it is necessary to be able to compose existing components into more complex objects, and to guarantee that this composition will work correctly and fulfill its expected role. Classical, static interface typing does not allow to reach this goal. Gathering teams specialized in behavioural specifications of components, languages and models for distributed, mobile, and communicating application programming, and methods and tools for compositional verification, the goal of FIACRE was to design methods and tools for specification, model extraction, and verification of distributed, hierarchical, and communicating components. The project work-plan was articulated around the following axes :

– Definition of a specification formalism for component behaviours, which must be adapted to verify distributed applications and allow an easy translation into the low-level formalisms that are used for verification.

– Development of semi-automated procedures for the behavioural model extraction of distributed components.
– Efficient tools for the verification (either using temporal logic formulas, behavioural equivalences, or behavioural typing) of the hierarchical compositions of components from their behavioural specifications.

In particular, within the collaborative project *Topcased*, and now supported by national RNTL platform *OpenEmbbed*, the FIACRE partners have defined an intermediate language for verification called Fiacre ("Format Intermédiaire pour les Architectures de Composants Répartis Embarqués") based on our developments, and that is the central exchange format for the verification tools of the OpenEmbbed platform [20].

The ReSeCo project

Type and Dates : Collaborative, Stic-Amsud, jan. 2007 - dec. 2009.
Title : Reliability and Security of Distributed Software Components
Partners : Univ. De la Republica (Montevideo, Uruguay) ; FAMAF, Univ. De Cordoba (Argentine) ; Univ. De Chili (Santiago, Chili) ; Univ. Diego Portales (Santiago, Chili)
My role : Participation, then Coordination
Abstract

The objective of the project ReSeCo (Reliability and Security of Distributed Software Components) is to investigate reliability and security in a computational model in which both the platform and applications are dynamic, so that incoming software, built from off-the-shelf components, may be destined to form part of the platform or to execute as a standard application. The concrete goals of the project include the development of mechanisms that help software developers build reliable software from of-the-shelf components, and of security infrastructures that guarantee end-users that the software they use is safe and secure .

The MCorePhP project

Type and Dates : ANR Blanc International, jan. 2010 - dec. 2012.
Title : Multi-Core Parallel Heterogeneous Programming
My role : Scientific and Management Participation
Partners : Tsinghua University (Pekin)
Abstract

In this MCorePHP project, we investigate certain methods and techniques that help simplify the parallel programming without sacrificing performance, in the main areas of scheduling, synchronization and proper use of the multi-core architecture features. Therefore, we need a safe, dependable, autonomic way of developing applications on multi-core processors, but also on multilevel infrastructures including multi-core, clusters, and large scale grid/cloud resources. The partners will ensure the compatibility of the new programming model with the China Grid specifications, and will assess the viability and efficiency of the approach on a large example from the area of bio-informatics.

At the semantic level, this project includes the development of a new programming model that contains information about the multilevel infrastructure, and provides users with a notion of multi-active object model. The idea is to allow some restricted form of sharing between activities that run in cores accessing a common memory, without unleashing the complexity of standard shared-memory

models. Sharing information comes in the form of user-defined annotations expressing the set of resources used by each method. This information is then used, together with information of the mapping of active object onto cores, statically or at run-time. This model will have some impact on the behavioural model used for verification : we need to modify and extend our models to take into account this new information, and the constraints on concurrency that are implied.

9.5 Participation to PhD juries

J.A. Montero de Queiroz, Univ. Paris 6, 1990 (*jury member*). Title : Représentations Graphiques, Transformations et Pré-Implémentation de LOTOS

Abderaman Lakas, Univ. Paris 6, 1996 (*jury member*). Title : Les Transformations Lotomaton : une contribution à la pré-implémentation des systèmes Lotos

Arnaud Février, ENST, janvier 1997 (*reviewer*). Title : Modèles et langages formels pour le point de vue de traitement ODP

Christophe Joubert, INPG, décembre 2005 (*jury member*). Title : Vérification distribuée à la volée de grands espaces d'états.

Jiri Adamek, Charles University, Prague, 2006 (*reviewer*). Title : Behaviour Composition in Component Systems

9.6 Activities as Students Adviser/Director

PhDs

1. **Didier Vergamini**

   PhD Thesis. Spécialité Informatique, Université de Nice
   Title : Vérification de Réseaux d'Automates Finis par Equivalences Observationnelles : le Système AUTO
   Defense : 1987, December 4th
   Adviser(s) : P. Franchi, E. Madelaine
   Reviewers : A. Arnold, G. Boudol
   Abstract

   To model parallel and communicating systems, we use process algebras introduced by R. Milner, allowing to give the semantics of terms representing such systems in form transition systems. In order to verify programs written in this formalism, we use the notion of observational equivalence attached to the notion of observational criterion. In order to implement this principle of verification, we built the system called AUTO, descended from the system ECRINS which allows the manipulation of process algebras. We give a full description of this system and of its implementation in LELISP. Its utilization is illustrated by very classic examples in the domain of the verification of parallel systems. After, we treat more complex examples of distributed algorithms.

2. **Rabéa Boulifa**

   PhD Thesis. Spécialité Informatique, Université de Nice Sophia-Antipolis
   Title : Génération de modèles comportementaux des applications réparties
   Defense : 2004, december 15th
   Adviser(s) : E. Madelaine

Reviewers : F. Vernadat, A. Fantechi
Abstract

In this work, we aim at developing automatic methods for verification of behavioural properties of distributed applications by methods based on models. Specifically, we study the question of building models from the source code of the Java applications. The models are labeled transition systems.

Therefore, we define a behavioural semantics for ProActive, a Java library for concurrent, distributed, and mobile computing. From this semantics we build behavioural models for finite abstractions of applications. These models are based on pro cess algebra semantics, so they can be built in a compositional manner. Building the finite models is not always possible. In order to deal the problems that take into account the data as well the problems concerning topologies with infinite objects, we define the notion of hierarchical models, based on parametrized transition systems and parametrized synchronization networks. By means of abstractions these models can depict infinite applications by expressive and finite representations. On the other hand, we define a system of semantics rules for building the (finite or parametrized) models from an intermediate form of programs obtained by static analysis. The models generated this way are used directly or after instantiation, standard by verification tools.

3. **Tomás Barros**

PhD Thesis. Spécialité Informatique. Université de Nice Sophia-Antipolis
Title : Formal Specification and Verification of Distributed Component Systems
Defense : 2005, November 25th
Adviser(s) : I. Attali, E. Madelaine
Reviewers : A.R. Cavalli, F. Plasil
Abstract

A component is a self contained entity that interacts with its environment through well-defined interfaces. The component library Fractive provides high level primitives and semantics for programming Java applications with distributed, asynchronous and hierarchical components. It also provides a separation between functional and non-functional aspects, the latter allows the execution control of a component and its dynamic evolution.

In this thesis, we provided a formal framework to ensure that the applications built from Fractive components are safe. Safe, in the sense that each component must be adequate to its assigned role within the system, and the update or replacement of a component should not cause deadlocks or failures to the system. We introduced a new intermediate format extending the networks of communicating automata, by adding parameters to their communication events and processes.

Then, we used this intermediate format to give behavioural specifications of Fractive applications. We assumed the models of the primitive components as known (given by the user or via static analysis). Using the component description, we built a controller describing the component's non-functional behaviour. The semantics of a component is then generated as the synchronization product of : its LTSs sub-components and the controller. The resulting system can be checked against requirements expressed in a set of temporal logic formulas, as illustrated in the thesis report.

4. **Antonio Cansado**

PhD Thesis. Spécialité Informatique. Université de Nice Sophia-Antipolis
Title : Formal Specification and Verification of Distributed Component Systems
Defense : 2008, October
Adviser(s) : E. Madelaine
Reviewers : F. Plasil, C. Canal
Abstract

Components are self-contained building blocks. They communicate through well-defined interfaces, that set some kind of contract. This contract must guarantee the behavioural compatibility of bound interfaces. This is particularly true when components are distributed and communicate through asynchronous method calls.

This thesis addresses the behavioural specification of distributed components. We develop a formal framework that allows us to build behavioural models. After abstraction, these models are a suitable input for state-of-the-art verification tools. The main objective is to specify, to verify, and to generate safe distributed components.

To this aim, we develop a specification language close to Java. This language is built on top of our behavioural model, and provides a powerful high-level abstraction of the system. The benefits are twofold : (i) we can interface with verification tools, so we are able to verify various kinds of properties ; and (ii), the specification is complete enough to generate code-skeletons defining the control part of the components. Finally, we validate our approach with a Point-Of-Sale case-study under the Common Component Model Example (CoCoME).

The specificities of the specification language proposed in this thesis are : to deal with hierarchical components that communicate by asynchronous method calls ; to give the component behaviour as a set of services ; and to provide semantics close to a programming language by dealing with abstractions of user-code.

Postdocs

1. **Régis Gascon**
   Title : Application of Formal Methods for the Verification of Infinite Systems to the Verification of Distributed Component Systems
   Dates : September 2008 to december 2009
   Adviser(s) : E. Madelaine
   Abstract

   In Vercors, the verification of safety properties on the formal models generated by the platform uses finite state model-checkers. As a consequence, this operation relies on the instantiation of some parameters and the finite abstraction or abstract interpretation of others. In order to improve this procedure, we have investigated the field of infinite state model-checking. This work follows three directions depending on the characteristics of pNets :
   – manipulation of “infinite data” (data in an infinite set),
   – representation of unbounded queues,
   – representation of parametrized topologies.
   Each of these points is a source of infinity since it introduces an unbounded parameter. The goal of infinite state model-checking methods is to consider directly these unbounded parameters without any abstraction. But the com-

bination of these different parameters makes the verification problem very difficult : from a theoretical point of view, any of the parameters mentioned above can imply undecidability of standard reachability analysis.

However, there are several techniques using semi-algorithm or restrictions of the general problems that allow tackling some problems in practice. We have studied some of these techniques with the objective to find a way to apply them to pNets. For instance, some representations using finite state automata can be used to represent (infinite) set of states in systems manipulating integers, queues or pointers. Regular expressions can also be used to describe the evolution of a parametrized or dynamic network during an execution. These are a few examples of the different possibilities that we have been studying. Now, an important open question is to understand how these techniques can be mixed together to verify safety properties as precisely as possible. Indeed, none of them is able to treat all the infinite aspects of Vercors formal model.

As experimentation, we have implemented a prototype to explore the set of configurations that can be reached in a network of finite state machines communicating with unbounded FIFO queues. We want to add mechanisms able to check liveness ; we would also like to define an input language to express the safety properties that can be checked using all our material. Of course, the other unbounded parameters linked to the data and the topology have to be taken into account. For the moment, such tools extensions are difficult to design since we do not exactly know how to mix the techniques needed with the current implementation. However, it is already possible to introduce some infinite data, with the condition that the set of their values can be finitely partitioned w.r.t. the kind of properties to verify.

Internships

This is a list of internships that I have proposed and directed from 2002 to 2010, in the Oasis team. They have been done by students from levels L3 to M2, or as engineer final internship.

- **Tomas Barros** : *Formalisation et preuves du système de factures électroniques au Chili*, Master, U. de Chile, 2002
- **Dao Anh Viet** : *Modèle comportemental pour calculs d'objets Répartis et Mobiles*, DEA, LIP6, 2002 [88]
- **Toufik Maarouk** : *Outils pour le model checking d'applications Java distribuées*, DEA U. d'Orléans, 2002 [67]
- **Christophe Massol** : *Outils d'analyse statique et de vérification pour les applications Java distribuées*, IUP Nice, 2003 [68]
- **Alejandro Vera** : *Formalisme graphique et éditeur pour réseaux de processus paramétrés*, Ingéniorat, U. de Chile, 2004
- **Walid Belkir** : *Spécifications comportementales de composants distribués*, M2, U. Aix-Marseille, 2005 [17]
- **Marcela Rivera** : *Efficient automatic tools for model-checking distributed Java applications*, M2, U. Técnica Federico Santa María, Chile. 2006
- **Hejer Rejeb** : *Construction par analyse statique de modèles comportementaux de composants distribués*, M2, LRI, 2006
- **Emil Salageanu** : *An UML Profile for the specification of distributed component systems*, EPU Nice, 2007 [83, 82]
- **Vivien Maisonneuve** : *Vérification de systèmes distribués utilisant une représentation des canaux à base d'automates*, L3, ENS Cachan, 2007

- **Pablo S. Valenzuela** : *Vercors Components Environment, an Eclipse plugin for Grid Component Model*, Ingéniorat, U. Diego Portales, Chili, 2008 [33]
- **Krzysztof Nirski** : *Une plateforme pour la spécification graphique de systèmes à base de composants distribués*, M1, AGH U. of Science and Technology, Krakovie, 2008
- **Mikolaj Baranowski** : *Une plateforme pour la spécification graphique de systèmes à base de composants distribués*, M1, AGH U. of Science and Technology, Krakovie, 2008
- **Adel Bouchakhchoukha** : *Génération de modèles comportementaux de composants GCM : formalisation et mise en oeuvre*, M2, U. Aix-Marseille, 2009 [26]
- **Amine Rouini** : *Topologies paramétrées pour les composants logiciels : extension et implantation d'un langage de description d'architecture*, M2, U. Nice-Sophia-Antipolis, 2010 [79]
- **Raluca Halalai** : *Packaging of CADP for ProActive Scheduling and Resourcing tools*, L3, U. Cluj, 2010

Chapitre 10

Personal Bibliography

Editions

[E-08] Eric Madelaine and Markus Lumpe, editors. *Proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS'07)*, volume vol 215. ENTCS, 2008.

[E-09] Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine, editors. *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, volume 5751 of *LNCS*. Springer, dec 2009.

Journals and Book Chapters

[J-92] E. Madelaine. Verification tools from the Concur project. *EATCS Bulletin*, 47, 1992.

[J-08] Antonio Cansado, Denis Caromel, Ludovic Henrio, Eric Madelaine, Marcela Rivera, and Emil Salageanu. *The Common Component Modeling Example : Comparing Software Component Models*, volume 5153 of *LNCS*, chapter A Specification Language for Distributed Components implemented in GCM/ProActive. Springer, 2008. `http://agrausch.informatik.uni-kl.de/CoCoME`.

[J-09] Tomás Barros, Rabéa Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, 64(1–2), jan 2009.

International Conferences with Selection Committee and Proceedings

[C-84] E. Madelaine. Un système d'aide à la preuve de compilateurs. In Manfred Paul and Bernard Robinet, editors, *International Symposium on Programming, 6th Colloquium, Toulouse, April 17-19, 1984, Proceedings*, volume 167 of *LNCS*. Springer, 1984.

[C-89] Eric Madelaine and Didier Vergamini. Auto : A verification tool for distributed systems using reduction of finite automata networks. In Son T. Vuong, editor, *Formal Description Techniques, II, Proceedings of the IFIP TC/WG6.1 : 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE 89, Vancouver, BC, Canada, 5-8 December, 1989*. North-Holland, 1989.

[C-90] Eric Madelaine and Didier Vergamini. Finiteness conditions and structural construction of automata for all process algebras. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV 90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*. Springer, 1990.

[C-91a] Eric Madelaine and Didier Vergamini. Specification and verification of a sliding window protocol in LOTOS. In *Formal Description Techniques, IV, Proceedings of the IFIP TC6/WG 6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE 91, Sydney, Australia, 19-22 November 1991*, volume C-2 of *IFIP Transactions*. North-Holland, 1991.

[C-91b] Eric Madelaine and Didier Vergamini. Tool demonstration : Tools for process algebras. In Ken R. Parker and Gordon A. Rose, editors, *Formal Description Techniques, IV, Proceedings of the IFIP TC6/WG 6.1 Fourth International Conference on Formal Description Techniques for Distrib uted Systems and Communication Protocols, FORTE 91, Sydney, Australia, 19-22 No vember 1991*, volume C-2 of *IFIP Transactions*. North-Holland, 1991.

[C-92] Eric Madelaine and Didier Vergamini. Verification of communicating processes by means of automata reduction and abstraction. In Alain Finkel and Matthias Jantzen, editors, *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, volume 577 of *Lecture Notes in Computer Science*. Springer, 1992.

[C-95] Rance Cleaveland, Eric Madelaine, and Steve Sims. A front-end generator for verification tools. In *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS 95, Aarhus, Denmark, May 19-20, 1995, Proceedings*, volume 1019 of *LNCS*. Springer, 1995.

[C-03a] R. Boulifa and E. Madelaine. Finite model generation for distributed java programs. In IEEE, editor, *Workshop on Model-Checking for Dependable software-Intensive Systems, DNS'03,*, pages 92–96, San Francisco, USA, Jan. 2003. IEEE.

[C-03b] R. Boulifa and E. Madelaine. Model generation for distributed Java programs. In E. Astesiano N. Guelfi and G. Reggio, editors, *Workshop on scientiFic engIneering of Distributed Java applIcations*, Luxembourg, nov 2003. Springer-Verlag, LNCS 2952.

[C-04a] Tomás Barros, Rabéa Boulifa, and Eric Madelaine. Parameterized models for distributed Java objects. In *Formal Techniques for Networked and Distributed Systems FORTE 2004*, volume LNCS 3235, pages 43–60, Madrid, September 2004. Spinger Verlag.

[C-04b] I. Attali, T. Barros, and E. Madelaine. Formalisation and proofs of the chilean electronic invoices system. In *in proc. of the XXIV International Conference of the Chilean Computer Science Society (SCCC'04)*, Arica, Chile, November 2004. IEEE.

[C-05a] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In Patrice Godefroid, editor, *Model Checking Software, 12th Int. SPIN Workshop*, San Francisco, CA, USA, August 2005. LNCS 3639, Springer.

[C-05b] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, volume 160, pages 41–55, Macao, October 2006. ENTCS.

[C-06] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model checking distributed components : The Vercors platform. In *3rd workshop on Formal*

Aspects of Component Systems (FACS'06), volume 182, pages 3–16, Prague, Czech Republic, Sep 2007. ENTCS.

[C-07a] S. Ahumada, L. Apvrille, T. Barros, A. Cansado, E. Madelaine, and E. Salageanu. Specifying Fractal and GCM Components With UML. In *proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC'07)*, Iquique, Chile, November 2007. IEEE.

[C-07b] Denis Caromel, Ludovic Henrio, and Eric Madelaine. Active objects and distributed components : Theory and implementation. In Frank S. de Boer and Marcello M. Bonsangue, editors, *FMCO 2007*, number 5382 in LNCS, pages 179–199, Berlin Heidelberg, 2008. Springer-Verlag.

[C-08a] Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Transparent first-class futures and distributed component. In *International Workshop on Formal Aspects of Component Software (FACS'08)*, volume 260, pages 155–171, Malaga, Sept 2008. Electronic Notes in Theoretical Computer Science (ENTCS).

[C-08b] Antonio Cansado, Eric Madelaine, and Pablo Valenzuela. VCE : A Graphical Tool for Architectural Definitions of GCM Components. Tool paper, 5th workshop on Formal Aspects of Component Systems (FACS'08), Sep 2008.

[C-08c] Antonio Cansado, Ludovic Henrio, Eric Madelaine, and Pablo Valenzuela. Unifying architectural and behavioural specifications of distributed components. In *International Workshop on Formal Aspects of Component Software (FACS'08)*, volume 260, pages 25–45, Malaga, Sept 2008. Electronic Notes in Theoretical Computer Science (ENTCS).

[C-09] Antonio Cansado and Eric Madelaine. Specification and verification for grid component-based applications : from models to tools. In *Formal Methods for Components and Objects (FMCO 2008)*, number 5751 in LNCS, pages 180–203, Berlin Heidelberg, 2009. Springer-Verlag.

[C-10] Rabéa Boulifa, Ludovic Henrio, and Eric Madelaine. Behavioural models for group communications. In *WCSI-10 : International Workshop on Component and Service Interoperability*, number 37 in EPTCS, pages 42–56, 2010.

Conferences without proceedings or local audience :

[W-04] T. Barros and E. Madelaine. Formal description and analysis for distributed systems. Technical Report 4-04, University of Kent, Computing Laboratory, April 2004. Doctoral Symposium at IFM'04, Canterbury, Kent, England.

[W-06] A. Cansado, L. Henrio and E. Madelaine. Towards Real-case Component Model-checking *In 5th Fractal Workshop* Nantes, France, July 2006

Thesis

[T-83] E. Madelaine. *Système d'aide à la preuve de compilateurs.* PhD thesis, Université de Paris VII, 1983.

Technical reports, Others

[R-82] E. Madelaine. *Le système Perluette et les preuves de reprsentation de types abstraits* Rapport de Recherche RR0133, INRIA, may 1982.

[R-85] M. Devin, A. Ressouche, E. Madelaine. *Application de CEYX a la construction de programmes sous forme de machines virtuelles* Rapport de Recherche RR0219, INRIA, may 1985.

[R-87a] V. Lecompte, D. Vergamini, E. Madelaine. *AUTO : un systeme de verification de processus paralleles et communicants* Rapport Technique RT083, INRIA, mar. 1987.

[R-87b] R. de Simone, E. Madelaine. *ECRINS : un laboratoire de preuve pour les calculs de processus* Rapport de Recherche RR672, INRIA, mar. 1987.

[R-88] G. Doumenc and E. Madelaine. *Une traduction de Plotos en Meije.* Rapport de Recherche RR938, INRIA, 1988.

[R-90] G. Doumenc, E. Madelaine, and R. de Simone. *Proving process calculi translations in ECRINS : The PureLotos --> Meije example* Rapport de recherche RR1192, INRIA, March 1990.

[R-92a] S. Gnesi, E. Madelaine, and G. Ristori. An exercise in protocol verification. In T. Bolognesi, E. Brinksma, and C. Vissers, editors, *Third Lotosphere Workshop and Seminar*, Pisa, September 1992.

[R-92b] E. Najm, A. Lakas, A. Serouchni, E. Madelaine, and R. de Simone. Alto : an interactive transformation tool for lotos and lotomaton. In T. Bolognesi, E. Brinksma, and C. Vissers, editors, *Third Lotosphere Workshop and Seminar*, Pisa, September 1992.

[R-95] W.R. Cleaveland, S. Sims and E. Madelaine. A front-end generator for verification tools Rapport de Recherche RR2612, INRIA, July 1995.

[R-02] Rabea Boulifa and Eric Madelaine. Preuves de propriétés de comportement de programmes proactive. Technical Report RR4460, INRIA, May 2002.

[R-04] T. Barros and E. Madelaine. Formalisation and proofs of the chilean electronic invoices system. Technical Report RR-5217, INRIA, june 2004.

[R-05a] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. Technical Report RR-5591, INRIA, June 2005.

[R-06] OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Proposals for a grid component model. Technical Report D.PM.02, CoreGRID, Programming Model Virtual Institute, Feb 2006. Responsible for the delivrable.

[R-07a] Yu Feng, Eric Madelaine, Ian Stockes-Rees and partners of the NESSI-Grid consortium. Grid Vision and Strategic Research Agenda. Technical report, NESSI-Grid : Networked European Software and Services Initiative - Grid, October 2007.

[R-07b] OASIS team and GridCOMP partners. Proceedings of the first gridcomp workshop. Technical report, GridComp, December 2007. Deliverable D.DIS.03.

[R-08a] Tomás Barros, Rabéa Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural models for distributed fractal components. Research Report 6491, INRIA, April 2008.

[R-08b] NESSI-Grid WP1 partners. Grid Vision and Strategic Agenda, Deliverable D.1.5. Technical report, NESSI-Grid SSA EU project, Nov 2008. final version.

[R-09] Régis Gascon and Eric Madelaine. Verifying distributed systems with unbounded channels. Technical report, INRIA, 2009.

[R-10] Ludovic Henrio and Eric Madelaine. Experiments with distributed model-checking of group-based applications. In *Sophia-Antipolis Formal Analysis Workshop*, page 3p., France Sophia-Antipolis, Oct 2010.

Standards, Software

[S-08a] ETSI TC-GRID. ETSI TS 102 827 : GRID ; Grid Component Model ; Part 1 : GCM Interoperability Deployment. Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2008.

[S-08b] ETSI TC-GRID. ETSI TS 102 828 : GRID ; Grid Component Model ; Part 2 : GCM Application Description. Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2008.

[S-09] ETSI TC-GRID. ETSI TS 102 829 : GRID ; Grid Component Model ; Part 3 : GCM Fractal Architecture Description Language (ADL). Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2009.

[S-10] ETSI TC-GRID. ETSI TS 102 830 : GRID ; Grid Component Model ; Part 4 : GCM Fractal JAVA API. Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2010.

[L-10] OASIS Team. The VERCORS platform : Verification of models for distributed communicating componants, with safety and security, 2010. `http://agrausch.informatik.uni-kl.de/CoCoME`.

[L-88] E. Madelaine, R. de Simone, and D. Vergamini. *ECRINS*, user manual, 1988. Technical Documentation.

Chapitre 11

General Bibliography

Bibliographie

[1] Apache Tuscany open source project. http ://tuscany.apache.org/..., ? ? ?

[2] SCOrWare : An Open SCA-compliant Service-oriented Component-based software platform. http ://www.scorware.org/, ? ? ?

[3] SOFA 2.0 : balancing advanced features in a hierarchical component model. In *Proceedings of SERA 2006, IEEE CS*, pages 40–48, Aug 2006.

[4] OASIS Committee Draft 5. SCA Assembly Specification Version 1.1. http ://oasis..., 2010.

[5] P. André, G. Ardourel, and C. Attiogbé. Adaptation for hierarchical components and services. *Electron. Notes Theor. Comput. Sci.*, 189 :5–20, 2007.

[6] P. André, G. Ardourel, and C. Attiogbé. Composing Components with Shared Services in the Kmelia Model. In *7th International Symposium on Software Composition, SC'08*, volume 4954 of *LNCS*. Springer, 2008.

[7] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE : A real-time uml profile supported by a formal validation toolkit. *IEEE transactions on software Engineering*, 30(7), 2004.

[8] A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.

[9] C. Attiogbé, P. André, and G. Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition (ETAPS/SC'06*, volume 4089 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[10] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[11] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing : Software Environments and Tools*, chapter 9 : Programming, Composing, Deploying for the Grid. Springer, 2006. ISBN : 1-85233-998-5.

[12] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 119–122. Springer Berlin / Heidelberg, 2010.

[13] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN Workshop*, LNCS 2057. Springer-Verlag, 2001.

[14] T. Barros. *Formal specification and verification of distributed component systems.* PhD thesis, University of Nice - Sophia Antipolis, November 2005.

[15] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. GCM : A Grid Extension to Fractal for Autonomous Distributed Components. *Annals of Telecommunications*, 64(1) :5–24, 2009.

[16] G. Behrmann, David A, K. G. Larsen, M. Oliver Möller, Paul Pettersson, and Wang Yi. UPPAAL - present and future. In *Proc. of 4*th *IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[17] W. Belhkir. Behavioural typing for proactice distributed components. Master thesis, Master recherche Univ. d'Aix-Marseille, June 2005.

[18] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[19] J.A. Bergstra, A. Pose, and S.A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.

[20] B. Berthomieu, J.P. Bodeveix, M. Filali, H. Garavel, F. La ng, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. The syntax and semantics of Fiacre. In *Rapport LAAS #07264 Rapport de Contrat Projet ANR05RNTL03101 OpenE mbeDD*, Mai 2007.

[21] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43 :1-50, 1996.

[22] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. *Formal Methods in System Design*, pages 237–255, 1999.

[23] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J.van Eijk, C.A.Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–76. North-Holland, 1989.

[24] A. Bouali, Annie R., V. Roy, and R. De Simone. The FCTOOLS User Manual (Version 1.0). Rapport Technique RT-0191, INRIA, 1996.

[25] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The fc2tools set. In *CAV*, pages 441–445, 1996.

[26] A. Bouchakhchoukha. Translation from pNets model to Fiacre language, for the verification of parallel, distributed and concurrent applications . Technical report, Master 2 MDFI, Univ. Aix-Marseille II, 2009.

[27] R. Boulifa. *Génération de modèles comportementaux des applications réparties*. PhD thesis, University of Nice - Sophia Antipolis – UFR Sciences, December 2004.

[28] P. Broadfoot and B. Roscoe. Tutorial on fdr and its applications. In *SPIN'00*, volume LNCS #1885, 2000.

[29] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12), 2006.

[30] J. R. Burch, E. M. Clarke, K. Mcmillan, D. Dill, and L. J. Hwang. Symbolic model checking : $10\hat{2}0$ states and beyond. In *Logic in Computer Science*, pages 428–439, 1990.

[31] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus : A tool for quantitative analysis of finite-state real-time systems. In *In ACM Workshop on Languages Compilers and Tools for Real-Time Systems*, 1995.

[32] A. Cansado. *Formal Specification and Verification of Distributed Component Systems*. PhD thesis, Université de Nice - Sophia Antipolis – UFR Sciences, December 2008.

[33] A. Cansado, E. Madelaine, and P. Valenzuela. VCE : A Graphical Tool for Architectural Definitions of GCM Components. 5th workshop on Formal Aspects of Component Systems (FACS'08), Sep 2008.

[34] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

[35] CCA forum. The Common Component Architecture (CCA) Forum home page, 2005. http ://www.cca-forum.org/.

[36] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. CALM and Cadena : Metamodeling for Component-Based Product-Line Development. *IEEE Computer*, 39(2), 2006.

[37] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counter-example-guided abstraction refinement. In *CAV'00*, volume LNCS #1855, pages 154–169, 2000.

[38] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[39] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[40] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *CONCUR'94*. LNCS 836, Springer, 1994.

[41] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from java source code. *Int. Conference on Software Engineering (ICSE)*, 2000.

[42] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.

[43] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, La Roche Posay, France, 1990. LNCS 469, Springer Verlag.

[44] A. Denis, C. Perez, T. Priol, and A. Ribes. Bringing high performance to the corba component model. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2004.

[45] M. Dwyer, J. Hatcliff, and H. Zheng. Slicing software for model construction. *Journal of High-order and Symbolic Computation*, 2000.

[46] F. Fernandes and J.C. Royer. The STSLIB project : Towards a formal component model based on STS. In *Proceedings of the Fourth International Workshop on Formal Aspects of Component Software (FACS'07)*, Sophia Antipolis, France, September 2007.

[47] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4 :13–24, aug 2002.

[48] H. Garavel, F. Lang, R. Mateescu, and W. Serve. Cadp 2010 : A toolbox for the construction and analysis of distributed processes. In *Proceedings of TACAS'11*, Saarbrcken, Germany, 2011. LNCS.

[49] H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm, and G. Stragier. Distributor and bcg_merge : Tools

for distributed explicit state space generation. In *TACAS'06*, pages 445–449, 2006.

[50] Servicio de impuestos internos Gobierno de Chile. Factura electrónica. Technical report, 2002.

[51] J.F. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, W. Wesselink, T. Willemse, and J. van der Wulp. The mCRL2 toolset. In *Proc. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT'08)*, 2008.

[52] J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mcrl2. In *Proc. Methods for Modelling Software Systems, Dagstuhl Seminar Proceedings 06351*, 2007.

[53] J.F. Groote and A. Ponse. Proof theory for μCRL : a language for processes with data. In Andrews et al., editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing Series, pages 231–250. Springer Verlag, 1994.

[54] O. Grumberg. Abstraction and refinement in model checking. In *FMCO'05*, volume LNCS #4111, pages 219–242, 2006.

[55] A. Gupta and O. Strichman. Abstraction refinement for bounded model checking. In *CAV'05*, volume LNCS #3576, pages 112–124, 2005.

[56] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad Ranganath. Cadena : An integrated development, analysis, and verification environment for component-based systems. In *ICSE'03*, 2006.

[57] Ludovic Henrio and Muhammad Uzair Khan. Asynchronous components with futures : Semantics and proofs in isabelle/hol. In *Proceedings of the Seventh International Workshop, FESCA 2010*. ENTCS, 2010.

[58] G. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6.

[59] G.J. Holzmann, American Telephone, and Telegraph Company. *Design and validation of computer protocols*. Prentice-Hall software series. Prentice Hall, 1991.

[60] P. Hošek, T. Pop, T. Bureš, P. Hntynka, and M. Malohlava. Comparison of component frameworks for real-time embedded systems. In Lars Grunske, Ralf Reussner, and Frantisek Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin / Heidelberg, 2010.

[61] G. Jung. The type system of CALM. Technical Report SAnToS-TR2006-3, Kansas State University, 2006.

[62] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 40, 1985.

[63] K. G. Larsen, P. Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, October 1997.

[64] F. Lerda, J. Kapinski, E. M. Clarke, and B. H. Krogh. Verification of supervisory control software using state proximity and merging. In *In Submitted to the 11th International Workshop on Hybrid Systems : Computation and Control*, 2008.

[65] H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 August 1996. LNCS 1119.

[66] G. Avrunin M. Dwyer and J. Corbett. Patterns in property specifications for finite-state verification. In *Int. Conf. on Software Engineering*, 1999.

[67] T. Maarouk. Outils pour le model-checking d'applications java distribuées. Master thesis, September 2002.

[68] C. Massols. Outils d'analyse statique et de vérification pour les applications java distribuées. Master thesis, UNSA, September 2003. Rapport de stage MIAGE, in french.

[69] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In S. Gnesi et al, editor, *Proceedings of FMICS'2000*, GMD Report 91, pages 65–86, Berlin, April 2000.

[70] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *FM'08*. LNCS 5014, 2008.

[71] R. Milner. Logic for computable functions : Description of a machine implementation. Technical report, Stanford University, 1972.

[72] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.

[73] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1), 1992.

[74] Jezek P., Kofron J., and Plàsil F. Model checking of component behavior specification : A real life experience. *ENTCS*, 160 :197–210, 2006.

[75] N. Parlavantzas, M. Morel, V. Getov, F. Baude, and D. Caromel. Performance and scalability of a component-based grid application. In *9th Int. Worshop on Java for Parallel and Distributed Computing, in conjunction with the IEEE IPDPS conference*, April 2007.

[76] L. C. Paulson. *Logic and computation. Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[77] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup : Architecture for component trading and dynamic updating. pages 43–52. IEEE CS Press, 1998.

[78] F. Ranzato. On the completeness of model checking. In *Proc. 10 th ESOP ' 2001 , Genova, IT, 2–6 Apr. 2001, LNCS 2028*, pages 137–154. Springer-Verlag, 2001.

[79] A. Rouini. Parametric Component Topologies : language extension and implementation. Technical report, Master Ubinet, Univ. of Nice Sophia Antipolis, 2010.

[80] V. Roy. *AUTOGRAPH : Un Outil de Visualisation pour les Calculs de Processus*. PhD thesis, 1990.

[81] E. Salageanu. An uml profile for the specification of distributed components systems. Technical report, Université de Nice-Sophia Antipolis, 2006. Rapport de stage de Master 1.

[82] E. Salageanu. Environment for the specification of distributed components. Technical report, Université de Nice-Sophia Antipolis, September 2007. Rapport de stage de Master 2.

[83] E. Salageanu. Tools for the behavioural modeling of distributed components. Technical report, Université de Nice-Sophia Antipolis, June 2007. Rapport de projet de fin d'etudes de Master 2.

[84] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, Oxford University Computing Laboratory, 1998.

[85] B. Thome and V. Viswanathan. Enterprise Grid Alliance–Reference Model v1.0. Apr. 2005.

[86] F. Tronel, F. Lang, and H. Garavel. Compositional verification using CADP of the ScalAgent deployment protocol for software components. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS'2003*, Paris, France, Nov 2003.

[87] D. Vergamini. *Vérification de Réseaux d'Automates finis par Équivalences observationnelles : le système AUTO.* PhD thesis, 1987.

[88] Dao Anh Viet. Modèle comportemental pour calculs d'objets répartis et mobiles. Master's thesis, Université Pierre et Marie Curie, LIP6, September 2002. Stage de DEA de Systèmes Informatiques Répartis.

[89] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proceedings of the 10th International Conference on Computer Aided Verification*, volume LNCS 1427 of *CAV '98*, pages 88–97. Springer-Verlag, 1998.